IBM XL Fortran for Linux, V16.1

Language Reference
for Little Endian Distributions

Version 16.1

IBM

IBM XL Fortran for Linux, V16.1

Language Reference
for Little Endian Distributions

Version 16.1

Contents

About this document

This document, which is part of the XL Fortran documentation suite, describes the syntax, semantics, and IBM implementation of the Fortran programming language on the Linux operating system. Although XL Fortran implementations conform to TS 29113 (partial support), Fortran 2008 (partial support), Fortran 2003 (full support), and other specifications maintained by the ISO standards for the Fortran programming language, they also incorporate many extensions to the core language. These extensions have been implemented to enhance usability in specific operating environments, assure compatibility with other compilers, and support new hardware capabilities.

Who should read this document

This document is a reference for users who already have experience programming in Fortran. Users new to Fortran can still use this document to find information about the language and features unique to XL Fortran; however, it does not aim to teach programming concepts nor to promote specific programming practices.

How to use this document

While this document covers both standard and implementation-specific features of XL Fortran, it does not include the following topics, which are covered in other documents:

- An executive overview of new functions: see the *What's New for XL Fortran*.
- Migration considerations: see the *XL Fortran Migration Guide*.
- Installation, system requirements, last-minute updates: see the *XL Fortran Installation Guide* and product README file.
- Overview of XL Fortran features: see *Getting Started with XL Fortran*.
- Compiler setup, compiling and running programs, compiler options, diagnostics: see the *XL Fortran Compiler Reference*.
- Optimizing, porting, OpenMP and SMP programming: see the *XL Fortran Optimization and Programming Guide*.
- Operating system commands related to the use of the compiler: consult your man page help and documentation of the Linux specific distribution.

How this document is organized

The following lists group information into sections that provide details on particular language topics and implementations:

- XL Fortran language elements:
  - XL Fortran for Linux
  - XL Fortran language fundamentals
  - Intrinsic data types
  - Derived types
  - Arrays concepts
  - Expressions and assignment
  - Execution control
  - Program units and procedures

- XL Fortran Input/Output
- Input/Output formatting
- Statements and attributes
- Directives (IBM extension)
- Intrinsic procedures
- The TRANSACTIONAL_MEMORY intrinsic module (IBM extension)
- Vector intrinsic procedures (IBM extension)
- Language interoperability features (Fortran 2003)
- The ISO_FORTRAN_ENV intrinsic module
- Extensions for source compatibility (IBM extension)
- Procedures that provide hardware-related functionality, and additional features for those who are already familiar with the Fortran language:
  - Floating-point control and inquiry procedures
  - Hardware–specific directives
  - Hardware–specific intrinsic procedures (IBM extension)
  - Service and utility procedures (IBM extension)
- The appendices provide information about compatibility across standards for users of earlier versions of Fortran and the ASCII and EBCDIC character sets mapping table.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM® XL Fortran for Linux, V16.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
lowercase bold	Invocation commands, executable names, and compiler options.	The compiler provides basic invocation commands, **xlf**, along with several other compiler invocation commands to support various Fortran language levels and compilation environments. The default file name for the executable program is **a.out**.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
monospace	Examples of program code, reference to program code, file names, path names, command strings, or user-defined names.	To compile and optimize myprogram.f, enter: xlf myprogram.f -03.

Table 1. Typographical conventions (continued)

Typeface	Indicates	Example
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures. Uppercase letters may also be used to indicate the minimum number of characters required to invoke a compiler option/suboption.	The **ASSERT** directive applies only to the **DO** loop immediately following the directive, and not to any nested **DO** loops.

Qualifying elements (icons and bracket separators)

In descriptions of language elements or programming models, this information uses icons and marked bracket separators to delineate segments of text as follows:

Table 2. Qualifying elements

Icon	Bracket separator text	Meaning
F2008 / F2008	Fortran 2008 begins / Fortran 2008 ends	The text describes an IBM XL Fortran implementation of the Fortran 2008 standard.[1]
F2003 / F2003	Fortran 2003 begins / Fortran 2003 ends	The text describes an IBM XL Fortran implementation of the Fortran 2003 standard, and it applies to all later standards.[1]
TS 29113 / TS 29113	TS 29113 begins / TS 29113 ends	The text describes an IBM XL Fortran implementation of Technical Specification 29113, referred to as TS 29113.[1]
IBM / IBM	IBM extension begins / IBM extension ends	The text describes a feature that is an IBM XL Fortran extension to the standard language specifications.
CUDA Fortran / CUDA Fortran	CUDA Fortran begins / CUDA Fortran ends	The text describes CUDA Fortran, the CUDA Fortran support provided by IBM XL Fortran, or both.
GPU / GPU	GPU begins / GPU ends	The text describes the information that is relevant to offloading computations to the NVIDIA GPUs.

Note:

1. If the information is marked with a Fortran language standard icon or bracket separators, it applies to this specific Fortran language standard and all later ones. Otherwise, it applies to all Fortran language standards.

Syntax diagrams

Throughout this information, diagrams illustrate XL Fortran syntax. This section helps you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a command, directive, or statement.

  The ──► symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ►── symbol indicates that a command, directive, or statement is continued from the previous line.

  The ──►◄ symbol indicates the end of a command, directive, or statement.

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |── symbol and end with the ──| symbol.

  IBM XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.

  Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.

- Required items are shown on the horizontal line (the main path):

  ►►──keyword──*required_argument*──────────────────────────────────►◄
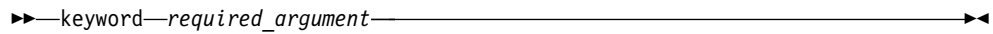
- Optional items are shown below the main path:

  ►►──keyword──┬──────────────────┬──────────────────────────────────►◄
               └─*optional_argument*─┘
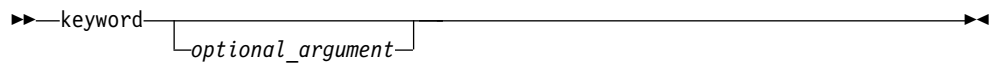
  Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([and]). For example, [UNIT=]u

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

  ►►──keyword──┬─*required_argument1*─┬──────────────────────────────►◄
               └─*required_argument2*─┘
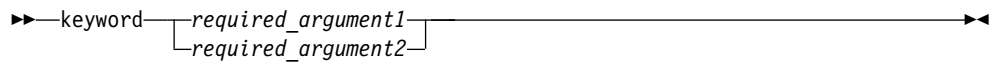
  If choosing one of the items is optional, the entire stack is shown below the main path.

  ►►──keyword──┬──────────────────────┬──────────────────────────────►◄
               ├─*optional_argument1*─┤
               └─*optional_argument2*─┘
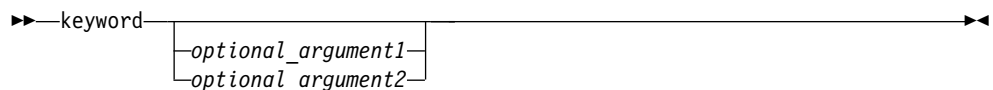
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
        ┌─,─────────────────┐
        │  ▼                 │
►►──keyword──repeatable_argument──────────────────────────────────►◄
```

- The item that is the default is shown above the main path.

```
              ┌─default_argument─┐
►►──keyword───┴─alternate_argument─────────────────────────────────►◄
```
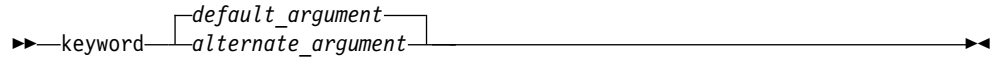
- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values. If a variable or user-specified name ends in _*list*, you can provide a list of these terms separated by commas.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following is an example of a syntax diagram with an interpretation:

```
                (1)                          ┌─,────┐
                                             │   ▼   │
►►──EXAMPLE──────char_constant──┬─a─┬──┬─────┴─e──┴──name_list──────►◄
                                └─b─┘  ├─c─┤
                                       └─d─┘
```

Notes:

1   IBM extension

Interpret the diagram as follows:
- Enter the keyword EXAMPLE.
- EXAMPLE is an IBM extension.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- 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. (The _*list* syntax is equivalent to the previous syntax for *e*.)

How to read syntax statements

Syntax statements are read from left to right:
- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:
- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b,* but not for both.
- Optionally, enter a value for *c* or *d.*
- Enter at least one value for *e.* If you enter more than one value, you must put a comma between each.
- Optionally, 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 *name.*

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

The examples for installation information are labelled as either *Example* or *Basic example. Basic examples* are intended to document a procedure as it would be performed during a default installation; these need little or no modification.

Notes on the terminology used

Some of the terminology in this information is shortened as follows:
- The term *free source form format* often appears as *free source form.*
- The term *fixed source form format* often appears as *fixed source form.*
- The term *XL Fortran* often appears as *XLF.*

Related information

The following sections provide related information for XL Fortran:

Available help information
IBM XL Fortran information

XL Fortran provides product information in the following formats:
- Quick Start Guide

  The Quick Start Guide (quickstart.pdf) is intended to get you started with IBM XL Fortran for Linux, V16.1. It is located by default in the XL Fortran directory and in the \quickstart directory of the installation DVD.
- README files

  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL Fortran directory, and in the root directory and subdirectories of the installation DVD.
- Installable man pages

Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL Fortran for Linux, V16.1 Installation Guide*.

- Online product documentation

  The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSAT4T_16.1.0/com.ibm.compilers.linux.doc/welcome.html.

- PDF documents

  PDF documents are available on the web at https://www.ibm.com/support/knowledgecenter/SSAT4T_16.1.0/com.ibm.compilers.linux.doc/download_pdf.html.

  The following files comprise the full set of XL Fortran product information.

  Note: To ensure that you can access cross-reference links to other XL Fortran PDF documents, download and unzip the .zip file that contains all the product documentation files, or you can download each document into the same directory on your local machine.

Table 3. XL Fortran PDF files

Document title	PDF file name	Description
What's New for IBM XL Fortran for Linux, V16.1, GC27-8043-00	`whats_new.pdf`	Provides an executive overview of new functions in the IBM XL Fortran for Linux, V16.1 compiler, with new functions categorized according to user benefits.
Getting Started with IBM XL Fortran for Linux, V16.1, GI13-3566-00	`getstart.pdf`	Contains an introduction to XL Fortran, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL Fortran for Linux, V16.1 Installation Guide, GC27-8040-00	`install.pdf`	Contains information for installing XL Fortran and configuring your environment for basic compilation and program execution.
IBM XL Fortran for Linux, V16.1 Migration Guide, GC27-8044-00	`migrate.pdf`	Contains migration considerations for using XL Fortran to compile programs that were previously compiled on different platforms, by previous releases of XL Fortran, or by other compilers.
IBM XL Fortran for Linux, V16.1 Compiler Reference, SC27-8050-00	`compiler.pdf`	Contains information about the various compiler options and environment variables.
IBM XL Fortran for Linux, V16.1 Language Reference, SC27-8048-00	`langref.pdf`	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to nonproprietary standards, compiler directives and intrinsic procedures.

Table 3. XL Fortran PDF files (continued)

Document title	PDF file name	Description
IBM XL Fortran for Linux, V16.1 Optimization and Programming Guide, SC27-8049-00	`proguide.pdf`	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.
Getting Started with CUDA Fortran programming using IBM XL Fortran for Linux, V16.1, GI13-3565-00	`getstart_cudaf.pdf`	Contains detailed information about the CUDA Fortran support that is provided in XL Fortran, including the compiler flow for CUDA Fortran programs, compilation commands, useful compiler options and macros, supported CUDA Fortran features, and limitations.

To read a PDF file, use Adobe Reader. If you do not have Adobe Reader, you can download it (subject to license terms) from the Adobe website at http://www.adobe.com.

More information related to XL Fortran, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036672.

For more information about the compiler, see the XL compiler on Power® community at http://ibm.biz/xl-power-compilers.

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

Standards and specifications

XL Fortran is designed to support the following standards and specifications. You can refer to these standards and specifications for precise definitions of some of the features found in this information.

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*.
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992*.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1*.
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991*. (This information uses its informal name, Fortran 90.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997*. (This information uses its informal name, Fortran 95.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004*. (This information uses its informal name, Fortran 2003.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2010*. (This information uses its informal name, Fortran 2008. We currently provide partial support to this standard.)
- *Information technology - Further interoperability of Fortran with C, ISO/IEC TS 29113:2012*. (This information uses its informal name, Technical specification 29113, referred to as TS 29113. We currently provide partial support to this specification.)

- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753* (United States of America, Department of Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.
- *OpenMP Application Program Interface Version 3.1* (full support), *OpenMP Application Program Interface Version 4.0 (partial support)*, and *OpenMP Application Program Interface Version 4.5 (partial support)*, available at http://www.openmp.org

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

Technical support

Additional technical support is available from the XL Fortran Support page at https://www.ibm.com/support/home/product/U128148Q26691I65/XL_Fortran_for_Linux. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send an email to compinfo@cn.ibm.com.

For the latest information about XL Fortran, visit the product information site at https://www.ibm.com/us-en/marketplace/xl-fortran-linux-compiler-power.

How to send your comments

Your feedback is important in helping us to provide accurate and high-quality information. If you have any comments about this information or any other XL Fortran information, send your comments to compinfo@cn.ibm.com.

Be sure to include the name of the manual, the part number of the manual, the version of XL Fortran, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. XL Fortran for Linux

The *XL Fortran Language Reference* is part of a documentation suite that offers information on installing and using the XL Fortran compiler on Linux. This document defines the syntax, semantics, and restrictions you must follow to write valid XL Fortran programs.

Fortran (FORmula TRANslation) is a high-level programming language primarily useful for engineering, mathematical, and scientific applications involving numeric computations.

XL Fortran implements partial Technical specification 29113, partial Fortran 2008, full Fortran 2003, and other language specifications maintained by the ISO standards for the Fortran programming language. XL Fortran also incorporates many extensions to the core language. These extensions have been implemented with the aims of enhancing usability in specific operating environments, assuring compatibility with other compilers, and supporting new hardware capabilities. A program that compiles correctly on one standard-conforming compiler should compile and execute correctly under all other conforming compilers, insofar as hardware differences permit.

The compiler detects most nonconformances to the XL Fortran language rules. The compiler cannot detect all combinations of syntax and semantic nonconformances because the diagnosis might hinder performance. XL Fortran programs that contain these undiagnosed nonconformances are not valid, even though they might run as expected.

Standards and specifications

XL Fortran is designed to support the following standards and specifications. You can refer to these standards and specifications for precise definitions of some of the features found in this information.

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*.
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992*.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1*.
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991*. (This information uses its informal name, Fortran 90.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997*. (This information uses its informal name, Fortran 95.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004*. (This information uses its informal name, Fortran 2003.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2010*. (This information uses its informal name, Fortran 2008. We currently provide partial support to this standard.)
- *Information technology - Further interoperability of Fortran with C, ISO/IEC TS 29113:2012*. (This information uses its informal name, Technical specification 29113, referred to as TS 29113. We currently provide partial support to this specification.)

- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753* (United States of America, Department of Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.
- *OpenMP Application Program Interface Version 3.1* (full support), *OpenMP Application Program Interface Version 4.0 (partial support)*, and *OpenMP Application Program Interface Version 4.5 (partial support)*, available at http://www.openmp.org

Technical specification 29113

An important supplement to Fortran 2008 is the ISO Technical Specification on Further Interoperability of Fortran with C, referred to as Technical specification 29113 or TS 29113. XL Fortran compiler supports the following subset of features in TS 29113:

- "Interoperability of assumed-length arguments (TS 29113)" on page 894
- "Interoperability of assumed-rank arguments (TS 29113)" on page 897
- "Interoperability of assumed-type arguments (TS 29113)" on page 895
- "Interoperability of allocatable and pointer arguments (TS 29113)" on page 889
- "Interoperability of optional arguments (TS 29113)" on page 889
- "Interoperable variables in asynchronous communication (TS 29113)" on page 886
- "The ISO_Fortran_binding.h header file (TS 29113)" on page 905

Fortran 2008

Segments of this document contain information based on the Fortran 2008 standard. The standard is open to continual interpretation, modification and revision. IBM reserves the right to modify the behavior of any features of this product to conform with future interpretations of this standard.

The Fortran Standards Committee responds to questions of interpretation about aspects of Fortran. Some questions can relate to language features already implemented in the XL Fortran compiler. Responses provided by the committee relating to these language features can result in changes to future releases of the XL Fortran compiler. These changes may result in incompatibilities with previous releases of the product.

Some of the Fortran 2008 features are as follows:

Module enhancements
- Submodules provide additional structuring facilities for modules.
- To declare a module procedure interface body or define a separate module procedure, specify the **MODULE** prefix specifier for the "FUNCTION" on page 379 or "SUBROUTINE" on page 468 statement.

Performance enhancements
- The DO CONCURRENT construct provides a means for programs to specify that individual loop iterations have no interdependencies.
- The CONTIGUOUS attribute provides a means for programs to specify restrictions on the storage layout of pointer targets and assumed-shape dummy arguments.

Data declaration

- A named constant array's shape can be implied by its value. For details, see Implied-shape arrays.
- A "FORALL" on page 371 index variable can have its type and kind explicitly declared within the construct.
- In addition to derived types, the **TYPE** type specifier is extended to declare entities of intrinsic types.
- Multiple type-bound procedures can be declared in a single type-bound procedure statement.

Data usage and computation
- You can allocate more than one *allocate_object* by using an **ALLOCATE** statement that contains the **SOURCE=** or **MOLD=** specifier.
- The **MOLD=** specifier has been added to the "ALLOCATE" on page 287 statement. In addition, you can omit the bounds in the **ALLOCATE** statement if you provide *source_expr* in the **SOURCE=** or **MOLD=** specifier.
- The real and imaginary parts of a complex entity can be accessed independently by using complex part designators.
- Variables now can be polymorphic in an intrinsic assignment.

Input and output
- **NEWUNIT=** in an OPEN statement automatically selects a unit number that does not interfere with other unit numbers that are selected by the program.

Execution control
- The BLOCK construct can contain declarations of objects with construct scope.
- The EXIT statement can transfer control from within more named executable constructs.
- The STOP statement can now take an integer or character constant expression as stop code. The ERROR STOP statement initiates error termination of a program while the **STOP** statement initiates normal termination of a program.

Intrinsic procedures
- The intrinsic functions ACOS, ASIN, ATAN, COSH, SINH, TAN, and TANH can have arguments of type complex.
- The new intrinsic functions ACOSH, ASINH, and ATANH calculate the inverse hyperbolic `cosine`, `sine`, and `tangent` respectively.
- The new intrinsic functions DSHIFTL and DSHIFTR calculate combined left and right shifts.
- The new intrinsic functions ERF, ERFC, and ERFC_SCALED calculate the error function and its complement.
- The EXECUTE_COMMAND_LINE subroutine can be used to pass a command to the operating system for execution.
- The new intrinsic function FINDLOC searches an array for a value.
- The new intrinsic functions GAMMA and LOG_GAMMA calculate the gamma function and its log.
- The new intrinsic function HYPOT calculates the Euclidean distance.
- The new intrinsic function IS_CONTIGUOUS(ARRAY) tests contiguity of an array.
- The new intrinsic functions LEADZ and TRAILZ return the number of leading and trailing zero bits in an integer.

- The new intrinsic functions MASKL and MASKR return simple left and right justified masks.
- A **BACK=** argument is added to the intrinsic functions MAXLOC and MINLOC.
- The new intrinsic functions POPCNT and POPPAR return the number of 1 bits of an integer and its parity.
- The new intrinsic functions SHIFTA, SHIFTL, and SHIFTR perform shift operations.
- A **RADIX=** argument has been added to the intrinsic procedure SELECTED REAL KIND.

Intrinsic modules
- The functions COMPILER_VERSION and COMPILER_OPTIONS in the ISO_FORTRAN_ENV intrinsic module return information about the program translation phase.
- The ISO_FORTRAN_ENV intrinsic module provides the following constants that are related to the Fortran environment: CHARACTER_KINDS, INT8, INT16, INT32, INT64, INTEGER_KINDS, IOSTAT_INQUIRE_INTERNAL_UNIT, LOGICAL_KINDS, REAL32, REAL64, REAL128, and REAL_KINDS.
- A **RADIX=** argument has been added to the IEEE_SELECTED_REAL_KIND intrinsic procedure.

Programs and procedures
- A dummy argument that has the **POINTER** and **INTENT(IN)** attributes can be argument that is associated with a nonpointer actual argument that has the **TARGET** attribute. For details, see Pointer dummy argument enhancements.
- Internal procedures and pointers to internal procedures can be used as actual arguments. Besides, internal procedures can be used as procedure pointer targets.
- If a dummy argument of an elemental procedure does not have the **VALUE** attribute, the dummy argument must have the **INTENT** attribute specified.
- In a reference to an elemental procedure, if any actual argument is an array, every actual argument that corresponds to an **INTENT(OUT)** or **INTENT(INOUT)** dummy argument must be an array.
- A separate module subprograms defines a separate module procedure that is declared by a corresponding module procedure interface body.
- Elemental procedures are no longer required to be pure in Fortran 2008. You can explicitly declare procedures with the **IMPURE** prefix specifier.
- The generic resolution rules are extended to distinguish between allocatable and pointer dummy arguments and between procedure and data dummy arguments. For details, see "Unambiguous generic procedure references" on page 168.
- A double colon separator (**::**) can be used in **PROCEDURE** and **MODULE PROCEDURE** statements inside interface blocks.
- The **FUNCTION** and **SUBROUTINE** keywords can be omitted from the END statement for a module or internal subprogram.
- You can specify the **BIND** attribute without the **NAME=** specifier for an internal procedure.
- You can specify the **VALUE** attribute on an array dummy argument that has either assumed shape or explicit shape.

Fortran 2003

Fortran 2003 offers many new features and feature enhancements to Fortran 95. Some of the major new features in Fortran 2003 are:

- Derived type enhancements
- Object-oriented programming support: type extension, type-bound procedures, type finalization, abstract and generic interfaces, polymorphism and PASS attribute
- Scoping and data manipulation enhancements: allocatable components, VOLATILE attribute, MAX, MIN, MAXLOC, MINLOC, MAXVAL and MINVAL intrinsics for character type
- Input/Output enhancements: User defined derived type I/O, asynchronous transfer including the WAIT statement
- Subroutine enhancements: VALUE attribute, Procedure pointers, deferred CHARACTER length
- Support for IEEE Floating Point Standard (IEEE 1989) exceptions
- Interoperability with the C programming language

Related information
- "Allocatable and pointer components" on page 52

Fortran 95

The Fortran 95 language standard is upward-compatible with the FORTRAN 77 and Fortran 90 language standards, excluding deleted features. Some of the improvements provided by the Fortran 95 standard are:

- Default initialization
- **ELEMENTAL** procedures
- The **FORALL** construct statement
- **POINTER** initialization
- **PURE** functions
- Specification expressions

Fortran 90

Fortran 90 offers many new features and feature enhancements to FORTRAN 77. The following topics outline some of the key features that Fortran 90 brings to the FORTRAN 77 language:

- Array enhancements
- Control construct enhancements
- Derived types
- Dynamic behavior
- Free source form
- Modules
- Parameterized data types
- Procedure enhancements
- Pointers

FORTRAN 77

FORTRAN 77 introduced new features and enhancements to FORTRAN 66, for more information see:

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

IBM extensions

An IBM extension generally modifies a rule or restriction from a given standards implementation. In this document, IBM extensions to the Fortran 2008, Fortran 2003, Fortran 95, and Fortran 90 standards are marked as indicated in the Conventions section under Conventions, Standards, and Documentation.

OpenMP API Version 4.5

The OpenMP API provides additional features which you can use to supplement the existing Technical specification 29113, Fortran 2008, Fortran 2003, and other language specifications.

The OpenMP Architecture Review Board (ARB) responds to questions of interpretation about aspects of the API. Some of these questions can relate to interface features implemented in this version of the XL Fortran compiler. Responses provided by this board relating to the interface can result in changes in future releases of the XL Fortran compiler. These changes may result in incompatibilities with previous releases of the product.

You can find information pertaining to the implementation of OpenMP API Version 4.5 in the following sections:
- **OpenMP environment variables** in the *XL Fortran Optimization and Programming Guide*
- **SMP Directives** in the *XL Fortran Optimization and Programming Guide*

Standards documents

XL Fortran is designed according to the standards listed in the Standards documents section. You can refer to these standards for precise definitions of some of the features found in this document.

Chapter 2. XL Fortran language fundamentals

This section describes the fundamental aspects of an XL Fortran application. Refer to the following headings for more information:

Characters

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

Table 4. The XL Fortran character set

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

The characters have an order known as a collating sequence, which is the arrangement of characters that determines their sequence order for such processes as sorting, merging, and comparing. XL Fortran uses American National Standard Code for Information Interchange (ASCII) to determine the ordinal sequence of characters. See "ASCII and EBCDIC character sets" on page 1054 for a complete listing of the ASCII character set.

White space refers to blanks and tabs. The significance of white space depends on the source format. See "Lines and source formats" on page 10 for details.

A lexical token is a sequence of characters with an indivisible interpretation that forms a building block of a program. A lexical token can be a keyword, name, literal constant (not of type complex), operator, label, delimiter, comma, equal sign, colon, semicolon, percent sign, ::, or =>.

Names

A name is a sequence of any or all of the following elements:

- Letters (A-Z, a-z)
- Digits (0-9)
- Underscores (_)
- **IBM** Dollar signs ($) **IBM**

The first character of a name must not be a digit.

In Fortran 2003, the maximum length of a name is 63 characters. In Fortran 90 and Fortran 95, the maximum length of a name is 31 characters.

IBM extension

In XL Fortran, the maximum length of a name is 250 characters. Although you can begin a name with an underscore, the Linux operating system as well as the XL Fortran compiler and libraries use reserved names that begin with underscores.

The compiler translates all letters in a source program into lowercase unless they are in a character context. Character context refers to characters within character literal constants, character-string edit descriptors, and Hollerith constants.

Note: When you specify the **-qmixed** compiler option, the compiler does not translate names to lowercase. For example, XL Fortran treats

```
ia Ia iA IA
```

the same by default, but treats lower and uppercase letters as distinct if you specify **-qmixed**.

End of IBM extension

A name can identify entities such as:

- A variable
- A named constant
- A procedure
- A derived type
- A construct
- A **CRITICAL** construct
- A program unit
- A common block
- A namelist group

Designators

A designator is a name that identifies a data object followed by zero or more selectors such as array element selectors, array section selectors, substring selectors, **F2008** complex part selectors **F2008** , and component selectors. A subobject designator identifies the following items:

- An array element
- An array section
- A character substring

- **F2008** A complex part **F2008**
- A structure component

Operators

In Fortran an expression is comprised of operands and operators. For a detailed description of Fortran operators, see "Operators and expressions" on page 105

Table 5. XL Fortran operators

Arithmetic	Logical
Character	Primary
General	Relational

Statements

A Fortran statement is a sequence of lexical tokens. Statements are used to form program units.

IBM The maximum length of a statement in XL Fortran is 65,000 bytes. **IBM**

See Statements and Attributes for more information on statements supported by XL Fortran.

Statement keywords

A statement keyword is part of the syntax of a statement. A sequence of characters is not reserved in all contexts. A statement keyword is interpreted as an entity name if the keyword is used in such a context.

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 scoping unit. In fixed source form, a statement label can appear anywhere in columns 1 through 5 of the initial line of the statement. In free source form, such column restrictions do not apply.

IBM XL Fortran ignores all characters that appear in columns 1 through 5 on fixed source form continuation lines. **IBM**

Giving the same label to more than one statement in a scoping unit causes ambiguity, and the compiler generates an error. White space and leading zeros are not significant in distinguishing between statement labels. You can label any statement, but a statement label reference can only refer to an executable statement or a **FORMAT** statement. The statement making the reference and the statement referenced must be in the same scoping unit for the reference to resolve.

Delimiters

Delimiters are pairs used to enclose syntactic lists. XL Fortran supports the following delimiters:
- Parentheses: (...)
- Slashes: /.../
- Array constructors: (/.../)

- F2003 Array constructors: [...] F2003

Lines and source formats

A line is a horizontal arrangement of characters. A column is a vertical arrangement of characters, where each character, or each byte of a multibyte character, in a given column shares the same horizontal line position.

IBM Because XL Fortran measures lines in bytes, these definitions apply only to lines containing single-byte characters. Each byte of a multibyte character occupies one column. IBM

The kinds of lines are:

Initial line	Is the first line of a statement.
Continuation line	Continues a statement beyond its initial line.
Comment line	Does not affect the executable program and can be used for documentation. The comment text continues to the end of a line. Although comment lines can follow one another, a comment line cannot be continued. A line of all white space or a zero-length line is a comment line without any text. Comment text can contain any characters allowed in a character context.
	If an initial line or continuation line is not continued, or if it is continued but not in a character context, an inline comment can be placed on the same line, to the right of any statement label, statement text, and continuation character that may be present. An exclamation mark (!) begins an inline comment.
Conditional compilation line	Indicates that the line should only be compiled if recognition of conditional compilation lines is enabled. A conditional compilation sentinel should appear on a conditional compilation line. For more information, see Conditional compilation.
Debug Line	Indicates that the line is for debugging code (for fixed source form only). In XL Fortran the letter D or X must be specified in column 1. For more information, see Debug lines.
Directive line	Provides instructions or information to the compiler in XL Fortran. For more information, see Comment form directives.

─────────────────────── **IBM extension** ───────────────────────

In XL Fortran source lines can be in fixed source form or free source form format. Use the **SOURCEFORM** directive to mix source formats within the same program unit. When you use the **f77** or **fort77** invocation command, fixed source form is the default. When you use **xlf** or **xlf_r** to compile the .f, .F, .f77, or .F77 files, fixed source form is also the default. When you use the **xlf90**, **xlf90_r**, **xlf95**, **xlf95_r**, **xlf2003**, **xlf2003_r**, **xlf2008**, **xlf2008_r**, or CUDA Fortran **xlcuf** CUDA Fortran invocation command, Fortran 90 free source form is the default.

See *Compiling XL Fortran Programs* in the *XL Fortran Compiler Reference* for details on invocation commands.

─────────────────────── **End of IBM extension** ───────────────────────

Fixed source form

IBM A fixed source form line is a sequence of 1 to 132 characters. The default line size is 72 characters. This is also the Fortran standard line size. You can change the default using the **-qfixed**=right_margin compiler option. In XL Fortran there is no limit to the number of continuation lines for a statement, but the statement cannot be longer than 65,000 bytes. Fortran 2003 limits the number of continuation lines to 255, while Fortran 95 limits the number of continuation lines to 19.

In fixed source form, columns beyond the right margin are not part of the line and you can use these columns for identification, sequencing, or any other purpose. IBM

Except within a character context, white space is insignificant. You can embed white space between and within lexical tokens, without affecting the way the compiler treats them.

IBM Tab formatting means that there is a tab character in columns 1 through 6 of an initial line in XL Fortran, which directs the compiler to interpret the next character as being in column 7. IBM

Requirements for lines and for items on those lines are:

- A comment line begins with a C, c, or an asterisk (*) in column 1, or is all white space. Comments can also follow an exclamation mark (!), except when the exclamation mark is in column 6 or in a character context.
- For an initial line without tab formatting:
  - Columns 1 through 5 contain either blanks, a statement label, IBM a D or an X in column 1 optionally followed by a statement label. IBM
  - Column 6 contains a blank or zero.
  - Columns 7 through to the right margin contain statement text, possibly followed by other statements or by an inline comment.
- IBM For an initial line with tab formatting in XL Fortran:
  - Columns 1 through 6 begin with either blanks, a statement label, or a D or an X in column 1, optionally followed by a statement label. You must follow this with a tab character.
  - If you specify the **-qxflag=oldtab** compiler option, all columns from the column immediately following the tab character through to the right margin contain statement text, possibly followed by other statements and by an inline comment.
  - If you do not specify **-qxflag=oldtab** compiler option, all columns from column 7, which corresponds to the character after the tab, to the right margin contain statement text, possibly followed by other statements and by an inline comment. IBM
- For a continuation line:
  - Column 1 must not contain C, c, or an asterisk. Columns 1 through 5 must not contain an exclamation mark as the leftmost nonblank character.

    IBM Column 1 can contain a D or an X which signifies a debug line in XL Fortran. Otherwise, these columns can contain any characters allowed in a character context; these characters are ignored. IBM
  - Column 6 must contain either a nonzero character or a nonwhite space character. The character in column 6 is the continuation character. Exclamation marks and semicolons are valid continuation characters.

- Columns 7 through to the right margin contain continued statement text, possibly followed by other statements and an inline comment.
- Neither the **END** statement or a statement whose initial line appears to be a program unit **END** statement can be continued.

F2008 A semicolon separates statements on a single source line, except when appearing in a character context, in a comment, or in columns 1 through 6. F2008 Two or more semicolon separators that are on the same line and are themselves separated by only white space or other semicolons are considered to be a single separator. A separator that is the last character on a line or before an inline comment is ignored. Statements following a semicolon on the same line cannot be labeled. Additional statements cannot follow a program unit **END** statement on the same line.

Debug lines (IBM extension)

A debug line, allowed only for fixed source form, contains source code used for debugging and is specified in XL Fortran by the letter D, or the letter X in column 1. The handling of debug lines depends on the **-qdlines** or the **-qxlines** compiler options:

- If you specify the **-qdlines** option, the compiler interprets the D in column 1 as a blank, and handles such lines as lines of source code. If you specify **-qxlines** , the compiler interprets the X in column 1 as a blank and treats these lines as source code.
- If you do not specify **-qdlines** or **-qxlines**, the compiler handles such lines as comment lines. This is the default setting.

If you continue a debugging statement on more than one line, every continuation line must have a continuation character as well as a D or an X in column 1. 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 **-qdlines** or **-qxlines** compiler option.

Example of fixed source form

```
C Column Numbers:
C        1         2         3         4         5         6         7
C234567890123456789012345678901234567890123456789012345678901234567890123456789012

!IBM* SOURCEFORM (FIXED)
      CHARACTER CHARSTR ; LOGICAL X        ! 2 statements on 1 line
      DO 10 I=1,10
        PRINT *,'this is the index',I  ! with an inline comment
10    CONTINUE
C
      CHARSTR="THIS IS A CONTINUED
     X CHARACTER STRING"
       ! There will be 38 blanks in the string between "CONTINUED"
       ! and "CHARACTER". You cannot have an inline comment on
       ! the initial line because it would be interpreted as part
       ! of CHARSTR (character context).
  100 PRINT *, IERROR
! The following debug lines are compiled as source lines if
! you use -qdlines
D     IF (I.EQ.IDEBUG.AND.
D    +    J.EQ.IDEBUG)     WRITE(6,*) IERROR
D     IF (I.EQ.
D    +  IDEBUG )
D    +  WRITE(6,*) INFO
      END
```

Free source form

A free source form line can specify up to 132 characters on each line. In XL Fortran, there is no limit to the number of continuation lines for a statement, but the statement cannot be longer than 65,000 bytes. Fortran 2003 limits the number of continuation lines to 255, while Fortran 95 limits the number of continuation lines to 39.

Items can begin in any column of a line, subject to the following requirements for lines and items on those lines:

- A comment line is a line of white space or begins with an exclamation mark that is not in a character context.
- An initial line can contain any of the following items, in the following sequence:
  - A statement label.
  - Statement text. Note that statement text is required in an initial line.
  - Additional statements.
  - The ampersand continuation character.
  - An inline comment.
- If you want to continue an initial line or continuation line in a non-character context, the continuation line must start on the first noncomment line that follows the intial line or continuation line. To define a line as a continuation line, you must place an ampersand after the statements on the previous non-comment line.
- White space before and after the ampersand is optional, with the following restrictions:
  - If you also place an ampersand in the first nonblank character position of the continuation line, the statement continues at the next character position following the ampersand.
  - If a lexical token is continued, the ampersand must immediately follow the initial part of the token, and the remainder of the token must immediately start after the ampersand on the continuation line.
- A character context can be continued if the following conditions are true:
  - The last character of the continued line is an ampersand and is not followed by an inline comment. If the rightmost character of the statement text to be continued is an ampersand, you must enter a second ampersand as a continuation character.
  - The first nonblank character of the next noncomment line is an ampersand.

F2008 A semicolon separates statements on a single source line, except when the semicolon appears in a character context or in a comment. F2008 Two or more separators that are on the same line and are themselves separated by only white space or other semicolons are considered to be a single separator. A separator that is the last character on a line or before an inline comment is ignored. Additional statements cannot follow a program unit **END** statement on the same line.

White space

White space must not appear within lexical tokens, except in a character context or in a format specification. You can freely insert white space between tokens to improve readability, and white space must separate names, constants, and labels from adjacent keywords, names, constants, and labels.

Certain adjacent keywords can require white space. The following table lists keywords where white space is optional.

Table 6. Keywords where white space is optional

BLOCK DATA	END FILE	END SUBMODULE **2**
DOUBLE COMPLEX **1**	END FORALL	END SUBROUTINE
DOUBLE PRECISION	END FUNCTION	END TYPE
ELSE IF	END IF	END UNION
ELSE WHERE	END INTERFACE	END WHERE
END ASSOCIATE	END MAP **1**	GO TO
END BLOCK **2**	END MODULE	IN OUT
END BLOCK DATA	END PROGRAM	SELECT CASE
END DO	END SELECT	SELECT TYPE **3**
END ENUM **3**	END STRUCTURE	
Note: **1** IBM extension **2** Fortran 2008 **3** Fortran 2003		

Example of free source form

```
!IBM* SOURCEFORM (FREE(F90))
!
! Column Numbers:
!        1         2         3         4         5         6         7
!234567890123456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,20
   PRINT *,'this statement&
   & is continued' ; IF (I.LT.5) PRINT *, I

 ENDDO
 EN&
        &D                  ! A lexical token can be continued
```

IBM free source form (IBM extension)

An IBM free source form line or statement is a sequence of up to 65,000 bytes. Items can begin in any column of a line, subject to the following requirements:

- A comment line begins with a double quotation mark in column 1, is a line of all white space, or is a zero-length line. A comment line must not follow a continued line. Comments can follow an exclamation mark except in a character context.
- An initial line can contain any of the following items, in the following sequence:
  - A statement label
  - Statement text
  - The minus sign continuation character
  - An inline comment
- A continuation line immediately follows a continued line and can contain any of the following items, in the following sequence:
  - Statement text
  - A continuation character
  - An inline comment

If statement text on an initial line or continuation line is to continue, a minus sign indicates continuation of the statement text on the next line. In a character context, if the rightmost character of the statement text to continue is a minus sign, a second minus sign must be entered as a continuation character.

Except within a character context, white space is insignificant. You can embed white space between and within lexical tokens, without affecting how the compiler treats those tokens.

Example of IBM free source form

```
!IBM* SOURCEFORM (FREE(IBM))
"
" Column Numbers:
"        1         2         3         4         5         6         7
"2345678901234567890123456789012345678901234567890123456789012
 DO I=1,10
  PRINT *,'this is -
            the index',I   ! There will be 14 blanks in the string
                           ! between "is" and "the"
 END DO
 END
```

Conditional compilation (IBM extension)

You can use sentinels to mark specific lines of an XL Fortran program for conditional compilation. This allows you to port code that contains statements that are only valid or applicable in an SMP environment to a non-SMP environment.

Conditional compilation is not supported with **IBM free source form**.

Syntax for conditional compilation

►►—*cond_comp_sentinel*—*fortran_source_line*————————————————————►◄
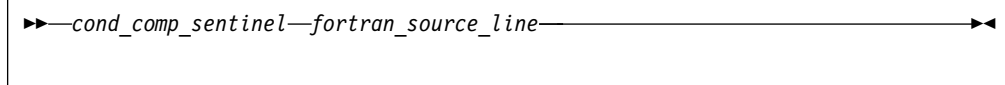
cond_comp_sentinel
      is a conditional compilation sentinel defined by the current source form and is either:
- **!$**, **C$**, **c$**, or ***$**, for fixed source form, or
- **!$**, for free source form

fortran_source_line
      is an XL Fortran source line

Conditional compilation rules

General rules:

A valid XL Fortran source line must follow the conditional compilation sentinel.

A conditional compilation line can contain the **EJECT**, **INCLUDE** or noncomment directives.

A conditional compilation sentinel must not contain embedded white space.

A conditional compilation sentinel must not follow a source statement or directive on the same line.

If you are continuing a conditional compilation line, the conditional compilation sentinel must appear on at least one of the continuation lines or on the initial line.

You must specify the **-qcclines** compiler option for conditional compilation lines to be recognized. To disable recognition of conditional compilation lines, specify the **-qnocclines** compiler option.

Trigger directives take precedence over conditional compilation sentinels. For example, if you specify the **-qdirective='$'** option, then lines that start with the trigger, such as **!$**, will be treated as comment directives, rather than conditional compilation lines.

Fixed source form rules:

Conditional compilation sentinels must start in column 1.

All rules for fixed source form line length, case sensitivity, white space, continuation, tab formatting, and columns apply.

Free source form rules:

Conditional compilation sentinels can start in any column.

All rules for free source form line length, case sensitivity, white space, and continuation apply. When you enable recognition of conditional compilation lines, two white spaces replace the conditional compilation sentinel.

Order of statements and execution sequence

In the *Statement order* table, vertical lines delineate statements that you can intersperse, while horizontal lines delineate statements that you cannot intersperse.

Table 7. Statement order

PROGRAM, FUNCTION, SUBROUTINE, MODULE, F2008 SUBMODULE F2008, or BLOCK DATA Statement		
USE Statements		
IMPORT Statements		
FORMAT and ENTRY Statements	IMPLICIT NONE	
	PARAMETER IMPLICIT statements	IMPLICIT statements
	PARAMETER and DATA statements	Derived-type definitions, interface blocks, type declaration statements, enumeration definitions, procedure declarations, specification statements, and statement function statements
	DATA statements	Executable constructs
CONTAINS Statement		
Internal subprograms or module subprograms		
END Statement		

Refer to Chapter 8, "Program units and procedures," on page 151 or Chapter 11, "Statements and attributes," on page 281 for more details on rules and restrictions concerning statement order.

The normal execution sequence is the processing of references to specification functions in any order, followed by the processing of executable statements in the order they appear in a scoping unit.

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 like **DO** and **IF**.
- Input/output statements like **READ** and **WRITE** that contain an **END=**, **ERR=**, or **EOR=** specifier.

When you reference a procedure that is defined by a subprogram, the execution of the program continues with any specification functions referenced in the scoping unit of the subprogram that defines the procedure. The program resumes with the first executable statement following the **ENTRY**, **FUNCTION** or **SUBROUTINE** statement that defines the procedure. When you return from the subprogram, execution of the program continues from the point at which the procedure was referenced or to a statement referenced by an alternate return specifier.

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

Data types

A data type consists of a name, a set of valid values, constants used as a way to denote those values, and a set of operations to manipulate those values. The two categories of data types are Intrinsic types and Derived types.

A derived type is a composite data type that can contain both intrinsic and derived data types.

Intrinsic types and their operations are predefined and always accessible. The two classes of intrinsic types are numeric and nonnumeric, with a number of types comprising each class.

Table 8. Intrinsic Types

Numeric Intrinsic Types	Nonnumeric Intrinsic Types
Integer	Logical
Real	Character
Complex	Vector **1**
Byte **1**	Byte **1**
Note: **1** IBM extension	

Type declaration: type parameters and specifiers

This is an overview section on declaring the type of an entity. The Statements section contains the particular syntax details and rules for derived and intrinsic type declarations.

XL Fortran provides one or more representation methods for each intrinsic data type. You can optionally specify this representation method with a kind type parameter value, using *kind_param* in your type declaration statement. This value can indicate:

- The range for the integer data type.
- The decimal precision and exponent range for the real data type.
- The decimal precision and exponent range for the complex data type.
- The representation method for the character data type.
- The representation method for the logical data type.

IBM The **BYTE** intrinsic type does not have a kind type parameter. **IBM**

A length type parameter specifies the number of characters for entities of type character.

A type specifier denotes the type of all entities declared in a type declaration statement. The **INTEGER**, **REAL**, **COMPLEX**, **LOGICAL**, and **CHARACTER** type specifiers can include a *kind_selector*, that specifies the kind type parameter.

For example, here are some common ways you can declare a 4-byte integer:

- **INTEGER(4)**
- **INTEGER(KIND=4)**
- **INTEGER**, where the default integer size is set to 4 bytes.
- **F2008** **TYPE(INTEGER(4))** **F2008**
- **F2008** **TYPE(INTEGER(KIND=4))** **F2008**
- **F2008** **TYPE(INTEGER)**, where the default integer size is set to 4 bytes. **F2008**

This document references 4-byte integers as **INTEGER(4)**.

See *Type Declaration* for detailed information about type specifiers.

Applicable intrinsic procedures

For objects of an intrinsic type, the **KIND** intrinsic procedure returns the kind type parameter of its argument.

You can use the **LEN** intrinsic procedure to determine the length type parameter of a character object.

IBM The **SIZEOF** intrinsic function returns the size of a data object in bytes. **IBM**

Type parameter inquiry

You can use a **type parameter inquiry** to identify the type parameter value of a data object.

Two examples of a type parameter inquiry are:

```
i%kind
```

```
string%len
```

Determining Type

Each user-defined function or named entity has a data type. The type of an entity accessed by host or use association is determined in the host scoping unit or accessed module, respectively. The type of a named entity is determined, in the following sequence, in one of three ways:

1. Explicitly, in one of the following ways:
   - From a specified type declaration statement (see "Type Declaration" on page 476 for details).
   - For function results, from a specified type statement or its **FUNCTION** statement.
2. Implicitly, from a specified **IMPLICIT** type statement.
3. 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 default integer. Otherwise, the type is default real.

In a given scoping unit, if a letter, dollar sign, or underscore has not been specified in an **IMPLICIT** statement, the implicit type used is the same as the implicit type used by the host scoping unit. A program unit and interface body are treated as if they had a host with an **IMPLICIT** statement listing the predefined conventions.

The data type of a literal constant is determined by its form.

Data objects

A data object is a variable, constant, or subobject of a constant.

A variable can have a value and can be defined or redefined during execution of an executable program. A variable can be:
- A scalar variable name
- An array variable name
- A subobject

A subobject of a variable is a portion of a named object that you can reference or define. A subobject can be:
- An array element.
- An array section
- A character substring
- A structure component

A subobject of a constant is a portion of a constant. The referenced portion can depend on a variable value.

Constants

A constant has a value and cannot be defined or redefined during execution of an executable program. A constant with a name is a named constant. You can use either the **ENUM** statement or the **PARAMETER** attribute to provide a constant with a name. A constant without a name is a literal constant. A literal constant can be of intrinsic type or typeless. A typeless constant can be:
- Hexadecimal
- Octal
- Binary
- Hollerith

The optional kind type parameter of a literal constant can only be a digit string or a scalar integer named constant.

A signed literal constant can have a leading plus or minus sign. All other literal constants must be unsigned. These constants do not have a leading sign. The value zero is neither positive nor negative. You can specify zero as signed or unsigned.

Automatic objects

An automatic object is a local data object dynamically allocated within a subprogram F2008 or a **BLOCK construct** F2008 . It can have a nonconstant character length, a nonconstant array bound, or both. An automatic object is not a dummy argument.

An automatic object always has the controlled automatic storage class.

You cannot specify an automatic object in any of the following statements:
- **COMMON**
- **DATA**
- **EQUIVALENCE**
- **NAMELIST**

Also, automatic objects cannot have the **AUTOMATIC**, **PARAMETER**, **SAVE**, or **STATIC** attributes. You cannot initialize or define an automatic object with a constant expression in a type declaration statement, but an automatic object can have a default initialization. An automatic object must not appear in the specification part of a main program, a module, F2008 or a submodule F2008 .

Polymorphic entities (Fortran 2003)

A polymorphic entity is a data entity that is able to be of differing types during program execution. The type of a data entity at a particular point during execution of a program is its dynamic type. The declared type of a data entity is the type that it is declared to have, either explicitly or implicitly.

You use the **CLASS** type specifier to declare polymorphic objects. If the **CLASS** type specifier contains a type name, the declared type of a polymorphic object is that type.

You can use the **CLASS(*)** specifier to declare an unlimited polymorphic object. An unlimited polymorphic entity is not declared to have a type. It is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

A nonpolymorphic entity is type-compatible only with entities of the same type. For a polymorphic entity, type compatibility is based on its declared type: a polymorphic entity that is not unlimited polymorphic is type-compatible with entities of the same type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type-compatible with all entities.

An entity is said to be type-compatible with a type if it is type-compatible with entities of that type. An entity is type-, kind-, and rank-compatible (TKR) with another entity if the first entity is type-compatible with the second, the kind type parameters of the first entity have the same values as corresponding kind type parameters of the second, and both entities have the same rank TS 29113 or either is assumed-rank TS 29113 .

Only components of the declared type of a polymorphic object may be designated by component selection.

A polymorphic allocatable object may be allocated to be of any type with which it is type-compatible. A polymorphic pointer or dummy argument may, during program execution, be associated with objects with which it is type-compatible.

The following table lists the dynamic type of objects.

Table 9. Dynamic type of objects

Object	Dynamic type
Allocated allocatable polymorphic object	The type with which the object was allocated.
Associated polymorphic pointer	The dynamic type of the pointer's target.
Nonallocatable nonpointer polymorphic dummy argument	The dynamic type of dummy's associated actual argument.
Deallocated allocatable	The allocatable object's declared type.
Disassociated pointer	The pointer's declared type.
Entity identified by an associate name	The dynamic type of the selector with which the object is associated.
Nonpolymorphic object	The object's declared type.

Related information
- "CLASS (Fortran 2003)" on page 312

Definition status of variables

A variable is either defined or undefined, and its definition status can change during program execution. A named constant has a value and cannot be defined or redefined during program execution.

Arrays (including sections), structures, and variables of character, complex or derived-type are objects made up of zero or more subobjects. Associations can be established between variables and subobjects and between subobjects of different variables.
- An object is defined if all of its subobjects are defined. That is, each object or subobject has a value that does not change until it becomes undefined or until it is redefined with a different value.
- A derived type scalar object is defined if and only if all of its nonpointer components are defined.
- A complex or character scalar object is defined if and only if all of its subobjects are defined.
- If an object is undefined, at least one of its subobjects is undefined. An undefined object or subobject cannot provide a predictable value.

Variables are initially defined if they are specified to have initial values by **DATA** statements, type declaration statements, or **STATIC** statements. Variables with the **BIND** attribute that are initialized by means other than Fortran are also initially defined. In addition, default initialization can cause a variable to be initially defined. Zero-sized arrays and zero-length character objects are always defined.

All other variables are initially undefined.

Events causing definition

The following events will cause a variable to become defined:

1. Execution of an intrinsic assignment statement other than a masked array assignment statement or **FORALL** assignment statement causes the variable that precedes the equal sign to become defined.

   Execution of a defined assignment statement may cause all or part of the variable that precedes the equal sign to become defined.

2. Execution of a masked array assignment statement or **FORALL** assignment statement may cause some or all of the array elements in the assignment statement to become defined.

3. As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data are transferred to it. Execution of a **WRITE** statement whose unit specifier identifies an internal file causes each record that is written to become defined.

   As execution of an asynchronous input statement proceeds, the variable does not become defined until the matching **WAIT** statement is executed.

4. Execution of a **DO** statement causes the **DO** variable, if any, to become defined.

5. Default initialization may cause a variable to be initially defined.

6. Beginning of execution of the action specified by an implied-**DO** list in an input/output statement causes the implied-**DO** variable to become defined.

7. Execution of an **ASSIGN** statement causes the variable in the statement to become defined with a statement label value.

8. A reference to a procedure causes the entire dummy argument data object to become defined if the dummy argument does not have **INTENT(OUT)**, and the entire corresponding actual argument is defined with a value that is not a statement label.

   A reference to a procedure causes a subobject of a dummy argument that does not have **INTENT(OUT)** to become defined if the corresponding subobject of the corresponding actual argument is defined.

9. Execution of an input/output statement containing an **IOSTAT=** specifier causes the specified integer variable to become defined.

10. F2003 Execution of an input/output statement containing an **IOMSG=** specifier causes the specified character variable to become defined when an error, end-of-file or end-of-record occurs. F2003

11. Execution of a **READ** statement containing a **SIZE=** specifier causes the specified integer variable to become defined.

12. Execution of a **READ** or **WRITE** statement in XL Fortran containing an **ID=** specifier causes the specified integer variable to become defined.

13. Execution of a **WAIT** statement in XL Fortran containing a **DONE=** specifier causes the specified logical variable to become defined.

14. Execution of a synchronous **READ** or **WRITE** statement in XL Fortran containing a **NUM=** specifier causes the specified integer variable to become defined.

    Execution of an asynchronous **READ** or **WRITE** statement containing a **NUM=** specifier does not cause the specified integer variable to become defined. The integer variable is defined upon execution of the matching **WAIT** statement.

15. Execution of an **INQUIRE** statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.

16. When a character storage unit becomes defined, all associated character storage units become defined.

    When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined, except that variables associated with the variable in an **ASSIGN** statement become undefined when the **ASSIGN** statement is executed. When an entity of type **DOUBLE PRECISION** becomes defined, all totally associated entities of double precision real type become defined.

    A nonpointer scalar object of type nondefault integer, real other than default or double precision, nondefault logical, nondefault complex, nondefault character of any length, or nonsequence type occupies a single unspecified storage unit that is different for each case. A pointer that is distinct from other pointers in at least one of type, kind, and rank occupies a single unspecified storage unit. When an unspecified storage unit becomes defined, all associated unspecified storage units become defined.

17. When a default complex entity becomes defined, all partially associated default real entities become defined.

18. When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.

19. When all components of a numeric sequence structure or character sequence structure become defined as a result of partially associated objects becoming defined, the structure becomes defined.

20. Execution of an **ALLOCATE** or **DEALLOCATE** statement with a **STAT=** specifier causes the variable specified by the **STAT=** specifier to become defined.

21. `F2003` If an error condition occurs during the execution of an **ALLOCATE** or **DEALLOCATE** statement that has an **ERRMSG=** specifier, the *errmsg-variable* becomes defined. `F2003`

22. Allocation of a zero-sized array causes the array to become defined.

23. Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.

24. Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.

25. Invocation of a procedure that contains a nonpointer, nonallocatable, automatic object, causes all nonpointer default-initialized subcomponents of the object to become defined.

26. Invocation of a procedure that contains a nonpointer nonallocatable **INTENT(OUT)** dummy argument causes all nonpointer default-initialized subcomponents of the object to become defined.

27. Allocation of an object of a derived type where a nonpointer component is initialized by default initialization, causes the component and its subobjects to become defined.

28. In a **FORALL** statement or construct used in Fortran 95, the *index-name* becomes defined when the *index-name* value set is evaluated.

29. `IBM` If a **THREADPRIVATE** nonpointer nonallocatable variable that does not appear in a **COPYIN** clause is defined on entry into the first parallel region, each new thread's copy of the variable is defined.

30. If a **THREADPRIVATE** common block that does not appear in a **COPYIN** clause is defined on entry into the first parallel region, each new thread's copy of the variable is defined.

31. For **THREADPRIVATE** variables that are specified in a **COPYIN** clause, each new thread duplicates the master thread's definition, allocation and association status of these variables. Therefore, if the master thread's copy of a variable is defined on entry to a parallel region, each new thread's copy of the variable will also be defined.

32. For **THREADPRIVATE** common blocks that are in a **COPYIN** clause, each new thread duplicates the master thread's definition, allocation and association status of the variables in these common blocks. Therefore, if the master thread's copy of a common block variable is defined on entry to a parallel region, each new thread's copy of the common block variable will also be defined.

33. When a variable is specified in a **FIRSTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **PARALLEL WORKSHARE**, **SECTIONS**, or **SINGLE** directive, each new thread duplicates the master thread's definition and association status of the variable. Therefore, if the master thread's copy of a variable is defined on entry to a parallel region, each new thread's copy of the variable will also be defined.

34. When a variable, a dummy argument, or a private variable that its data-sharing attribute is firstprivate in a **TASK** region, each task duplicates the definition of the generating task and the association of the variable. If the generating task's copy of a variable is defined on entry to the **TASK** region, each new task's copy of the variable is also defined.

35. For each variable, or variable inside a common block, specified in a **COPYPRIVATE** clause, then after the execution of the code enclosed in the **SINGLE** construct and before any threads in the team have left the construct, all copies of the variable become defined as follows:

    • If the variable has the **POINTER** attribute, then copies of the variable in other threads in the team have the same pointer association status as the copy of the variable belonging to the thread that executed the code enclosed in the **SINGLE** construct.

    • If the variable does not have the **POINTER** attribute, then copies of the variable in other threads in the team have the same definition as the copy of the variable belonging to the thread that executed the code enclosed in the **SINGLE** construct. `IBM`

36. `F2008` Successful execution of an **OPEN** statement containing a **NEWUNIT=** specifier causes the variable specified by the **NEWUNIT=** specifier to become defined. `F2008`

37. `F2008` For an unsaved, nonpointer, nonallocatable, local variable of a **BLOCK** construct, the execution of the **BLOCK** statement of the construct containing the variable causes all nonpointer, default-initialized ultimate components of the variable to become defined. `F2008`

38. An object with the **VOLATILE** attribute might become defined when the definition is not specified in the program.

Events causing undefinition

The following events will cause a variable to become undefined:

1. When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default complex, definition of one does not cause the other to become undefined.

2. Execution of an **ASSIGN** statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.

3. If the evaluation of a function may cause an argument of the function or a variable in a module, `F2008` in a submodule `F2008` , or in a common block to become defined, and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, the argument or variable becomes undefined when the expression is evaluated.

4. The execution of a **RETURN** statement or **END** statement within a subprogram causes all variables that are local to its scoping unit, or that are local to the current instance of its scoping unit for a recursive invocation, to become undefined, except for the following:

   a. Variables with the **SAVE** or **STATIC** attribute.

   b. Variables in blank common.

   c. According to Fortran 90, variables in a named common block that appears in the subprogram and appears in at least one other scoping unit that is making either a direct or indirect reference to the subprogram. `IBM` XL Fortran does not undefine these variables, unless they are part of a threadlocal common block. `IBM`

   d. Variables accessed from the host scoping unit.

   e. According to Fortran 90, variables accessed from a module that also is referenced directly or indirectly by at least one other scoping unit that is making either a direct or indirect reference to the subprogram. `IBM` XL Fortran does not undefine these variables. `IBM`

   f. According to Fortran 90, variables in a named common block that are initially defined and that have not been subsequently defined or redefined. `IBM` XL Fortran does not undefine these variables. `IBM`

5. When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list or namelist-group of the statement become undefined.

6. When an error condition, end-of-file condition, or end-of-record condition occurs during execution of an input/output statement and the statement contains any implied-**DO** lists, all of the implied-**DO** variables in the statement become undefined.

7. Execution of a defined assignment statement may leave all or part of the variable that precedes the equal sign undefined.

8. Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.

9. Execution of an **INQUIRE** statement may cause the **NAME=**, **RECL=**, **NEXTREC=**, and **POS=** variables to become undefined.

10. When a character storage unit becomes undefined, all associated character storage units become undefined.

    When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type (see (1) above).

    When an entity of double precision real type becomes undefined, all totally associated entities of double precision real type become undefined.

    When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.

11. A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part of the actual argument is defined with a value that is a statement label value.

12. When an allocatable entity is deallocated, it becomes undefined. Successful execution of an **ALLOCATE** statement for a nonzero-sized object for which default initialization has not been specified causes the object to become undefined.

13. Execution of an **INQUIRE** statement causes all inquiry specifier variables to become undefined if an error condition exists, except for the variable in the **IOSTAT=** or F2003 **IOMSG=** F2003 specifier, if any.

14. When a procedure is invoked:

    a. An optional dummy argument that is not associated with an actual argument is undefined.

    b. A nonpointer dummy argument with **INTENT(OUT)** and its associated actual argument are undefined, except for nonpointer direct components that have default initialization.

    c. A pointer dummy argument with **INTENT(OUT)** and its associated actual argument have an association status of undefined.

    d. A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined.

    e. The function result variable is undefined, unless it was declared with the **STATIC** attribute and was defined in a previous invocation.

15. When the association status of a pointer becomes undefined or disassociated, the pointer becomes undefined.

16. When the execution of a **FORALL** statement or construct in Fortran 95 has completed, the *index-name* becomes undefined.

17. F2003 When execution of a **RETURN** or **END** statement causes a variable to become undefined, any variable of type **C_PTR** becomes undefined if its value is the C address of any part of the variable that becomes undefined.

18. When a variable with the **TARGET** attribute is deallocated, any variable of type **C_PTR** becomes undefined if its value is the C address of any part of the variable that is deallocated. F2003

19. IBM When a variable is specified in either the **PRIVATE** or **LASTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **PARALLEL WORKSHARE**, **SECTIONS** or **SINGLE** directive, each new thread's copy of the variable is undefined when the thread is first created.

20. When a variable is specified in the **PRIVATE** clause of a **TASK** directive, each private copy of the variable is undefined when the task is first generated.

21. When a variable is specified in a **FIRSTPRIVATE** clause of a **PARALLEL**, **PARALLEL DO**, **DO**, **PARALLEL SECTIONS**, **PARALLEL WORKSHARE**, **SECTIONS**, **SINGLE** or **TASK** directive, each new thread duplicates the master thread's definition and association status of the variable. Therefore, if the master thread's copy of a variable is undefined on entry to a parallel region, each new thread's copy of the variable will also be undefined.

22. When a variable is specified in the **NEW** clause of an **INDEPENDENT** directive, the variable is undefined at the beginning of every iteration of the following **DO** loop.

23. When a variable appears in asynchronous input, that variable becomes undefined, and remains undefined, until the matching **WAIT** statement is reached.

24. If a **THREADPRIVATE** common block or a **THREADPRIVATE** variable is specified in a **COPYIN** clause, each new thread duplicates the master thread's definition, allocation and association status of the variables. Therefore, if the master thread's copy of a variable is undefined on entry to a parallel region, each new thread's copy of the variable will also be undefined.

25. **F2003** If a **THREADPRIVATE** common block variable or a **THREADPRIVATE** variable has the **ALLOCATABLE** attribute, the allocation status of each copy created will be not allocated. **F2003**

26. If a **THREADPRIVATE** common block variable or a **THREADPRIVATE** variable has the **POINTER** attribute with an initial association status of disassociated through either default or explicit initialization, each copy will have an association status of disassociated. Otherwise the association status of each copy is undefined.

27. If a **THREADPRIVATE** common block variable or a **THREADPRIVATE** variable has neither the **ALLOCATABLE** nor the **POINTER** attribute and is initially defined through default or explicit initialization, each copy has the same definition. Otherwise, each copy is undefined. **IBM**

28. **F2008** When execution of a **BLOCK** construct is complete:

    a. An unsaved, local variable of the **BLOCK** construct becomes undefined.

    b. A variable of type C_PTR becomes undefined if its value is the C address of an unsaved, local variable of the **BLOCK** construct.

    F2008

29. An object with the **VOLATILE** attribute might become undefined when the undefinition is not specified in the program.

Syntactic contexts of definition or undefinition of variables

The appearance of a variable in the following contexts implies definition or undefinition of the variable:

1. The variable of an assignment statement
2. A *pointer_object* in a **NULLIFY** statement
3. A data pointer object or procedure pointer object in a pointer assignment statement
4. A **DO** variable in a **DO** statement or an input/output statement containing implied-DO lists
5. An input item in a **READ** statement
6. A variable whose name is in a namelist statement if the namelist group name appears in a **NML**= specifier in a **READ** statement
7. An internal file variable in a **WRITE** statement
8. An **IOSTAT**=, **SIZE**=, or **IOMSG**= specifier in an input/output statement
9. A specifier in an **INQUIRE** statement except **FILE**=, **ID**=, and **UNIT**=
10. A **NEWUNIT**= specifier in an **OPEN** statement
11. A *stat_variable*, *allocate_object*, or *errmsg_variable*
12. An actual argument in a reference to a procedure with an explicit interface if the corresponding dummy argument has the **INTENT (OUT)** or **INTENT (INOUT)** attributes
13. A variable that is the selector in a **SELECT TYPE** or **ASSOCIATE** construct if the associate name of the construct appears in any of the variable definition contexts

Allocation status

The allocation status of an allocatable object is one of the following during program execution:

- Not allocated, which means that the object has never been allocated, if it is given that status by the allocation transfer procedure, or that the last operation on it was a deallocation.
- Allocated, which means that the object has been allocated by an **ALLOCATE** statement, if it is allocated during assignment, or if it is given that status by the allocation transfer procedure.
- **IBM** Undefined, which means that the object does not have the **SAVE** or **STATIC** attribute and was allocated when execution of a **RETURN** or **END** statement resulted in no executing scoping units having access to it. In XL Fortran, undefined status is only available when you use the **-qxlf90=noautodealloc** option. **IBM**

If the allocation status of an allocatable object is allocated, the object may be referenced and defined. An allocatable object that is not allocated must not be referenced or defined. If the allocation status of an allocatable object is undefined, the object must not be referenced, defined, allocated, or deallocated.

When the allocation status of an allocatable object changes, the allocation status of any associated allocatable object changes accordingly. Allocation of an allocatable variable establishes values for the deferred type parameters of all associated allocatable variables.

In the Fortran standard, the allocation status of an allocatable object that is declared in the scope of a module is processor-dependent if it does not have the **SAVE** attribute and was allocated when execution of a **RETURN** or **END** statement resulted in no executing scoping units referencing the module.

F2008 An unsaved, allocatable, local variable of a **BLOCK** construct is deallocated when execution exits the **BLOCK** construct.**F2008**

IBM In XL Fortran, the allocation status of such an object remains allocated.**IBM**

Storage classes for variables (IBM extension)

Note: This section pertains only to storage for variables. Named constants and their subobjects have a storage class of *literal*.

Fundamental storage classes

All variables are ultimately represented by one of five storage classes:

Automatic
for variables in a procedure that will not be retained once the procedure ends. Variables reside in the stack storage area.

Static for variables that retain memory throughout the program. Variables reside in the data storage area. Uninitialized variables reside in the bss storage area.

Common
for common block variables. If a common block variable is initialized, the whole block resides in the data storage area; otherwise, the whole block resides in the bss storage area.

Controlled Automatic
for automatic objects. Variables reside in the stack storage area. XL Fortran allocates storage on entry to the procedure and deallocates the storage when the procedure completes.

Controlled
for allocatable objects. Variables reside in the heap storage area. You must explicitly allocate and deallocate the storage.

Secondary storage classes

None of the following storage classes own their own storage, but are associated with a fundamental storage class at run time.

Pointee
is dependent on the value of the corresponding integer pointer.

Reference parameter
is a dummy argument whose actual argument is passed to a procedure using the default passing method or **%REF**.

Value parameter
is a dummy argument whose actual argument is passed by value to a procedure.

For details on passing methods, see "%VAL and %REF (IBM extension)" on page 196.

Storage class assignment

Variable names are assigned storage classes in one of the following ways:

1. Explicitly:
   - Dummy arguments have an explicit storage class of reference parameter or value parameter. See "%VAL and %REF (IBM extension)" on page 196 for more details.
   - Pointee variables have an explicit storage class of pointee.
   - Variables for which the **STATIC** attribute is explicitly specified have an explicit storage class of static.
   - Variables for which the **AUTOMATIC** attribute is explicitly specified have an explicit storage class of automatic.
   - Variables that appear in a **COMMON** block have an explicit storage class of common.
   - Variables for which the **SAVE** attribute is explicitly specified have an explicit storage class of static, unless they also appear in a **COMMON** statement, in which case their storage class is common.
   - Variables that appear in a **DATA** statement or are initialized in a type declaration statement have an explicit storage class of static, unless they also appear in a **COMMON** statement, in which case their storage class is common.
   - Function result variables that are of type character or derived have the explicit storage class of reference parameter.
   - Function result variables that do not have the **SAVE** or **STATIC** attribute have an explicit storage class of automatic.
   - Automatic objects have an explicit storage class of controlled automatic.
   - Allocatable objects have an explicit storage class of controlled.

   A variable that does not satisfy any of the above, but that is equivalenced with a variable that has an explicit storage class, inherits that explicit storage class.

A variable that does not satisfy any of the above, and is not equivalenced with a variable that has an explicit storage class, has an explicit storage class of static if:

- A **SAVE** statement with no list exists in the scoping unit or,
- The variable is declared in the specification part of a main program.

2. Implicitly:

If a variable does not have an explicit storage class, it can be assigned an implicit storage class as follows:

- Variables whose names begin with a letter, dollar sign or underscore that appears in an **IMPLICIT STATIC** statement have a storage class of static.
- Variables whose names begin with a letter, dollar sign or underscore that appears in an **IMPLICIT AUTOMATIC** statement have a storage class of automatic.

In a given scoping unit, if a letter, dollar sign or underscore has not been specified in an **IMPLICIT STATIC** or **IMPLICIT AUTOMATIC** statement, the implicit storage class is the same as that in the host.

Variables declared in the specification part of a module F2008 or submodule F2008 are associated with the static storage class.

A variable that does not satisfy any of the above but that is equivalenced with a variable that has an implicit storage class, inherits that implicit storage class.

3. Default:

All other variables have the default storage class:

- Static, if you specified the **-qsave=all** compiler option.
- Static, for variables of derived type that have default initialization specified, and automatic otherwise if you specify the **–qsave=defaultinit** compiler option.
- Automatic, if you specified the **-qnosave** compiler option. This is the default setting.

  See **-qsave** option in the *XL Fortran Compiler Reference* for details on the default settings with regard to the invocation commands.

Typeless literal constants

A typeless constant does not have an intrinsic type in XL Fortran. Hexadecimal, octal, binary, and Hollerith constants can be used in any situation where intrinsic literal constants are used, except as the length specification in a type declaration statement (although typeless constants can be used in a *type_param_value* in **CHARACTER** type declaration statements). The number of digits recognized in a hexadecimal, octal, or binary constant depends on the context in which the constant is used.

Hexadecimal constants

The form of a hexadecimal constant is:

```
>>------+--X--+--+-'--hexadecimal_number--'-+---------------------------><
        |  Z  |  +-"--hexadecimal_number--"-+
        +--'--hexadecimal_number--'--+--+--X--+--
        |  +-"--hexadecimal_number--"-+  +  Z  +
        +--Z--hexadecimal_number--------
```
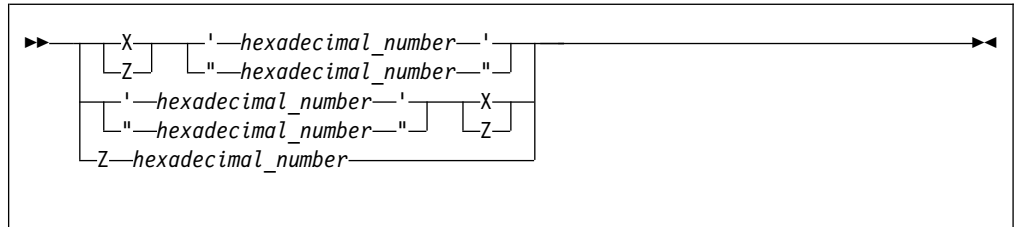
hexadecimal_number
> is a string composed of digits (0-9) and letters (A-F, a-f). Corresponding uppercase and lowercase letters are equivalent.

The **Z***nn...nn* form of a hexadecimal constant can only be used as a data initialization value delimited by slashes. If this form of a hexadecimal constant is the same string as the name of a constant you defined previously with the **PARAMETER** attribute, XL Fortran recognizes the string as the named constant.

If 2x hexadecimal digits are present, x bytes are represented.

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

Related information

See "Using typeless constants" on page 33 for information on how XL Fortran interprets the constant.

Octal constants

The form of an octal constant is:

```
>>---+--O--+--'--octal_number--'-+---------------------------------------><
     |     +-"--octal_number--"-+
     +--'--octal_number--'--+--O-+
     +-"--octal_number--"---
```
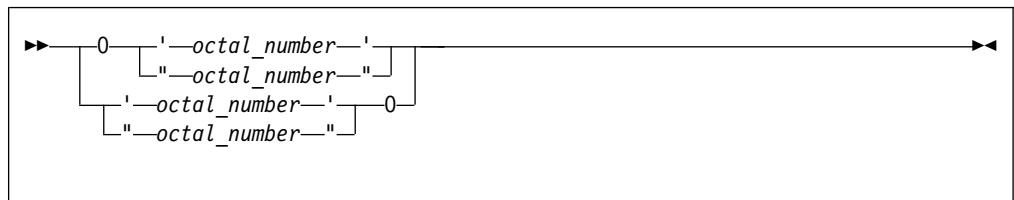
octal_number
> is a string composed of digits (0-7)

Because an octal digit represents 3 bits, and a data object represents a multiple of 8 bits, the octal constant may contain more bits than are needed by the data object. For example, an **INTEGER(2)** data object can be represented by a 6-digit octal constant if the leftmost digit is 0 or 1; an **INTEGER(4)** data object can be represented by an 11-digit constant if the leftmost digit is 0, 1, 2, or 3; an **INTEGER(8)** can be represented by a 22-digit constant if the leftmost digit is 0 or 1.

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

Related information

See "Using typeless constants" on page 33 for information on how the constant is interpreted by XL Fortran.

Binary constants

The form of a binary constant is:

```
►►─┬─B─┬─'─binary_number─'─┬─────────────────────────────────►◄
   │    └─"─binary_number─"─┘
   └─┬─'─binary_number─'─┬─B─┘
     └─"─binary_number─"─┘
```
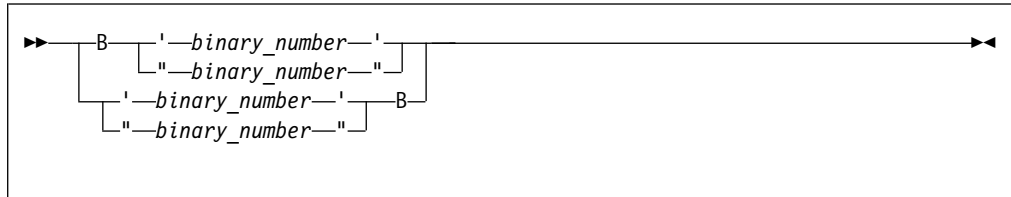
binary_number
        is a string formed from the digits 0 and 1

If 8x binary digits are present, x bytes are represented.

Examples

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

Related information

See "Using typeless constants" on page 33 for information on how XL Fortran interprets the constant.

Hollerith constants

The form of a Hollerith constant is:

```
►►─n─H─character_string────────────────────────────────────►◄
```
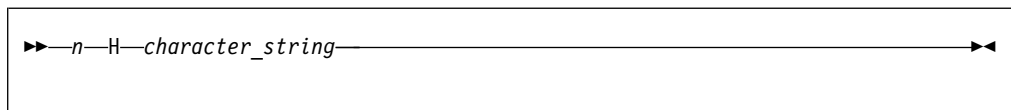
A Hollerith constant consists of a nonempty string of characters capable of representation in the processor and preceded by nH, where n is a positive unsigned integer constant representing the number of characters after the H. n cannot specify a kind type parameter. The number of characters in the string may be from 1 to 255.

Note: If you specify nH and fewer than n characters are specified after the n, any blanks that are used to extend the input line to the right margin are considered to be part of the Hollerith constant. A Hollerith constant can be continued on a continuation line. At least n characters must be available for the Hollerith constant.

XL Fortran also recognizes escape sequences in Hollerith constants, unless the **-qnoescape** compiler option is specified. If a Hollerith constant contains an escape sequence, n is the number of characters in the internal representation of the string, not the number of characters in the source string. (For example, 2H\"\" represents a Hollerith constant for two double quotation marks.)

XL Fortran provides support for multibyte characters within character constants, Hollerith constants, **H** edit descriptors, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.

Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

See "Using typeless constants" for information on how the constant is interpreted by XL Fortran.

Using typeless constants

The data type and length of a typeless constant are determined by the context in which you use the typeless constant. XL Fortran does not convert the data type and length until you use them and context is understood.

- If you compile your program with the **-qctyplss** compiler option, character constant expressions follow the rules that apply to Hollerith constants.
- A typeless constant can assume only one of the intrinsic data types.
- When you use a typeless constant with an arithmetic or logical unary operator, the constant assumes a default integer type.
- When you use a typeless constant with an arithmetic, logical, or relational binary operator, the constant assumes the same data type as the other operand. If both operands are typeless constants, they assume a type of default integer unless both operands of a relational operator are Hollerith constants. In this case, they both assume a character data type.
- When you use a typeless constant in a concatenation operation, the constant assumes a character data type.
- When you use a typeless constant as the expression on the right-hand side of an assignment statement, the constant assumes the type of the variable on the left-hand side.
- When you use a typeless constant in a context that requires a specific data type, the constant assumes that data type.
- When you use a typeless constant as an initial value in a **DATA** statement, **STATIC** statement, or type declaration statement, or as the constant value of a named constant in a **PARAMETER** statement, or when the typeless constant is to be treated as any noncharacter type of data, the following rules apply:
  - If a hexadecimal, octal, or binary constant is smaller than the length expected, XL Fortran adds zeros on the left. If it is longer, the compiler truncates on the left.
  - If a Hollerith constant is smaller than the length expected, the compiler adds blanks on the right. If it is longer, the compiler truncates on the right.
  - If a typeless constant specifies the value of a named constant with a character data type having inherited length, the named constant has a length equal to the number of bytes specified by the typeless constant.
- When a typeless constant is treated as an object of type character (except when used as an initial value in a **DATA**, **STATIC**, type declaration, or component definition statement), the length is determined by the number of bytes represented by the typeless constant.

- When you use a typeless constant as part of a complex constant, the constant 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 a typeless constant as an actual argument, the type of the corresponding dummy argument must be an intrinsic data type. The dummy argument must not be a procedure, pointer, array, object of derived type, or alternate return specifier.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure,
  - All of the arguments are typeless constants, and
  - There *is* a specific intrinsic procedure that has the same name as the generic procedure name,

  the reference to the generic name will be resolved through the specific procedure.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure,
  - All of the arguments are typeless constants, and
  - There is *no* specific intrinsic procedure that has the same name as the generic procedure name,

  the typeless constant is converted to default integer. If a specific intrinsic function takes integer arguments, the reference is resolved through that specific function. If there are no specific intrinsic functions, the reference is resolved through the generic function.
- When you use a typeless constant as an actual argument, and:
  - The procedure reference is to a generic intrinsic procedure, and
  - There is another argument specified that is not a typeless constant,

  the typeless constant assumes the type of that argument. However, if you specify the compiler option **-qport=typlssarg**, the actual argument is converted to default integer. The selected specific intrinsic procedure is based on that type.
- When you use a typeless constant as an actual argument, and the procedure name is established to be generic but is not an intrinsic procedure, the generic procedure reference must resolve to only one specific procedure. The constant assumes the data type of the corresponding dummy argument of that specific procedure. See Example 2.
- When you use a typeless constant as an actual argument, and the procedure name is established to be only specific, the constant assumes the data type of the corresponding dummy argument.
- When you use a typeless constant as an actual argument, and:
  - The procedure name has not been established to be either generic or specific, and
  - The constant has been passed by reference,

  the constant assumes the default integer size but no data type, unless it is a Hollerith constant. The default for passing a Hollerith constant is the same as if it were a character actual argument. However, using the compiler option **-qctyplss=arg** will cause a Hollerith constant to be passed as if it were an integer actual argument. See "Resolution of procedure references" on page 206 for more information about establishing a procedure name to be generic or specific.

- When you use a typeless constant as an actual argument, and:
  - The procedure name has not been established to be either generic or specific, and
  - The constant has been passed by value,

  the constant is passed as if it were a default integer for hexadecimal, binary, and octal constants.

  If the constant is a Hollerith constant and it is smaller than the size of a default integer, XL Fortran adds blanks on the right. If the constant is a Hollerith constant and it is larger than 8 bytes, XL Fortran truncates the rightmost Hollerith characters. See "Resolution of procedure references" on page 206 for more information about establishing a procedure name to be generic or specific.
- When you use a typeless constant in any other context, the constant assumes the default integer type, with the exception of Hollerith constants. Hollerith constants assume a character data type in the following situations:
  - An H edit descriptor
  - A relational operation with both operands being Hollerith constants
  - An input/output list
- If a typeless constant is to be treated as a default integer but the value cannot be represented within the value range for a default integer, the constant is promoted to a representable kind.
- A kind type parameter must not be replaced with a logical constant even if **-qintlog** is on, nor by a character constant even if **-qctyplss** is on, nor can it be a typeless constant.

Examples

Example 1

```
INT=B'1'          ! Binary constant is default integer
RL4=X'1'          ! Hexadecimal constant is default real
INT=INT + O'1'    ! Octal constant is default integer
RL4=INT + B'1'    ! Binary constant is default integer
INT=RL4 + Z'1'    ! Hexadecimal constant is default real
ARRAY(O'1')=1.0   ! Octal constant is default integer

LOGICAL(8) LOG8
LOG8=B'1'         ! Binary constant is LOGICAL(8), LOG8 is .TRUE.
```

Example 2

```
INTERFACE SUB
  SUBROUTINE SUB1( A )
     REAL A
  END SUBROUTINE
  SUBROUTINE SUB2( A, B )
     REAL A, B
  END SUBROUTINE
  SUBROUTINE SUB3( I )
     INTEGER I
  END SUBROUTINE
END INTERFACE
CALL SUB('C0600000'X, '40066666'X)  ! Resolves to SUB2

CALL SUB('00000000'X)               ! Invalid - ambiguous, may
                                    ! resolve to either SUB1 or SUB3
```

Chapter 3. Intrinsic data types

Intrinsic types and their operations are predefined and always accessible. The two classes of intrinsic types are numeric and nonnumeric, with a number of types comprising each class.

Table 10. Intrinsic Types

Numeric Intrinsic Types	Nonnumeric Intrinsic Types
Integer	Logical
Real	Character
Complex	Vector **1**
Byte **1**	Byte **1**
Note:	
1 IBM extension	

XL Fortran also supports derived types, which are composite data types that can contain both intrinsic and derived types.

Integer

─────────────────────────── **IBM extension** ───────────────────────────

The *Range of integer values* table contains the range of values that XL Fortran can represent using the integer data type.

Table 11. Range of integer values

Kind parameter	Range of values
1	-128 through 127
2	-32 768 through 32 767
4	-2 147 483 648 through 2 147 483 647
8	-9 223 372 036 854 775 808 through 9 223 372 036 854 775 807

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for integer values. Use the **-qintsize** compiler option to change the default integer size to 2, 4, or 8 bytes. Note that the **-qintsize** option similarly affects the default logical size.

──────────────────────── **End of IBM extension** ────────────────────────

The integer type specifier must include the **INTEGER** keyword.

The form of a signed integer literal constant is:
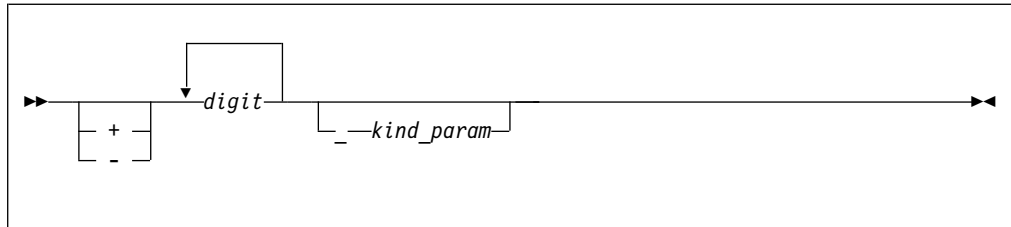
kind_param
> is either a *digit-string* or a *scalar-int-constant-name*

A signed integer literal constant has an optional sign, followed by a string of decimal digits containing no decimal point and expressing a whole number, optionally followed by a kind type parameter. A signed, integer literal constant can be positive, zero, or negative. If unsigned and nonzero, the constant is assumed to be positive.

If *kind_param* is specified, the magnitude of the literal constant must be representable within the value range permitted by that *kind_param*.

IBM extension

If no *kind_param* is specified in XL Fortran, and the magnitude of the constant cannot be represented as a default integer, the constant is promoted to a kind in which it can be represented.

XL Fortran represents integers internally in two's-complement notation, where the leftmost bit is the sign of the number.

End of IBM extension

Example of integer constants
```
0                       ! has default integer size
-173_2                  ! 2-byte constant
9223372036854775807     ! Kind type parameter is promoted to 8
```

Real

IBM extension

The following table shows the range of values that XL Fortran can represent with the real data type:

Kind Parameter	Approximate Absolute Nonzero Minimum	Approximate Absolute Maximum	Approximate Precision (decimal digits)
4	1.175494E-38	3.402823E+38	7
8	2.225074D-308	1.797693D+308	15
16	2.225074Q-308	1.797693Q+308	31

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for real values. Use the **-qrealsize** compiler option to change the default real size to 4 or 8 bytes. Note that the **-qrealsize** option affects the default complex size.

XL Fortran represents **REAL(4)** and **REAL(8)** 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). The **REAL(16)** representation is based on the **REAL(8)** format.

```
REAL(4)
Bit no. 0....|....1....|....2....|....3.
        seeeeeeeefffffffffffffffffffffff

REAL(8)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffffff

REAL(16)
Bit no. 0....|....1....|....2....|....3....|....4....|....5....|....6...
        seeeeeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Bit no. .|....7....|....8....|....9....|....0....|....1....|....2....|..
        seeeeeeeeeeefffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

This ANSI/IEEE binary floating-point format also provides representations for +infinity, -infinity, and NaN (not-a-number) values. A NaN can be further classified as a quiet NaN or a signaling NaN. See Implementation details of XL Fortran floating-point processing for details on the internal representation of NaN values.

The definition of intrinsic `RANGE` is `INT(MIN(LOG10(HUGE(X)), -LOG10(TINY(X))))`.

For **REAL(8)** numbers, the **HUGE** intrinsic returns `0x7FEFFFFFFFFFFFFF` and the **TINY** intrinsic returns `0x0010000000000000`. As a result, we have `INT(MIN(308.254715559916747, 307.652655568588784))`, and therefore the range is 307. Note that the LOG scale is not symmetric on both ends of the exponent.

The IBM format of **REAL(16)** numbers is composed of two **REAL(8)** numbers of different magnitudes that do not overlap. That is, the binary exponents differ by at least the number of fraction bits in a **REAL(8)**.

For **REAL(16)**, the **RANGE** intrinsic returns the range of the numbers that have both **REAL(8)** numbers normalized. Consequently, for **REAL(16)** numbers, the **HUGE** intrinsic returns `0x7FEFFFFFFFFFFFFF7C9FFFFFFFFFFFFF` and the TINY intrinsic returns `0x0360000000000000000000000000000000`. As a result, we have `INT(MIN(308.25471555991674389886862819788120, 291.69806579839777816211298898803388))`, where the range is 291.

308 is the lowest or highest exponent that can be represented in the **REAL(8)** or **REAL(16)** numbers.

———————————————————— **End of IBM extension** ————————————————————

A real type specifier must include either the **REAL** keyword or the **DOUBLE PRECISION** keyword. The precision of **DOUBLE PRECISION** values is twice that of default real values. See "REAL" on page 449 and "DOUBLE PRECISION" on page 344 for details on declaring entities of type real.

The forms of a real literal constant are:
- A basic real constant optionally followed by a kind type parameter
- A basic real constant followed by an exponent and an optional kind type parameter
- An integer constant (with no *kind_param*) followed by an exponent and an optional kind type parameter

A basic real constant has, in order, an optional sign, an integer part, a decimal point, and a fractional part. Both the integer part and 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 XL Fortran will use to approximate the value of the constant. XL Fortran interprets a basic real constant as a decimal number.

The form of a real constant is:

```
          ┌──────┐
          ▼      │
     ┌─┐   digit─┘  exponent──────────────────────────────►
     │+│
     └─┘
     ┌─┐   ┌──────┐
     │+│   ▼      │
     └─┘   digit─┘ . ┌──────┐  ┌──exponent──┐
                     ▼      │
                     digit─┘
     ┌─┐          ┌──────┐
     │+│        . ▼      │
     └─┘        digit─┘  ┌──exponent──┐
          ┌──────┐
          ▼      │
          digit─┘

►──┬─────────────────┬──►◄
   └─ _ ─kind_param──┘
```

exponent

```
►►─┬─E─┬─┬───┬─ digit_string ─────────────►◄
   ├─D─┤ │ + │
   └─Q─┘ └ - ┘
```
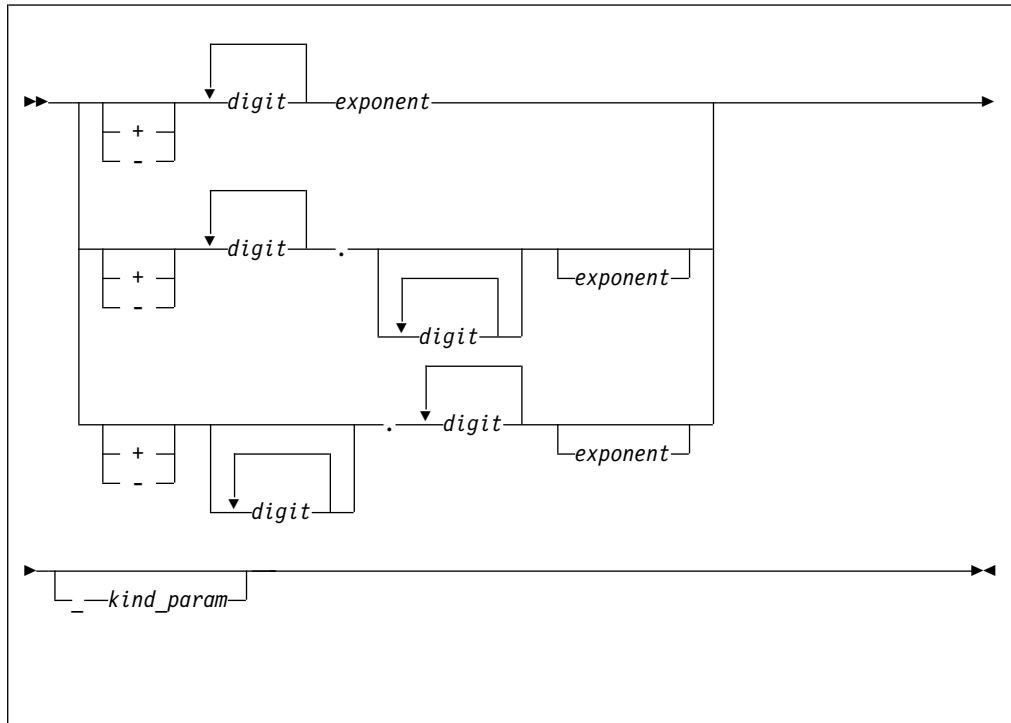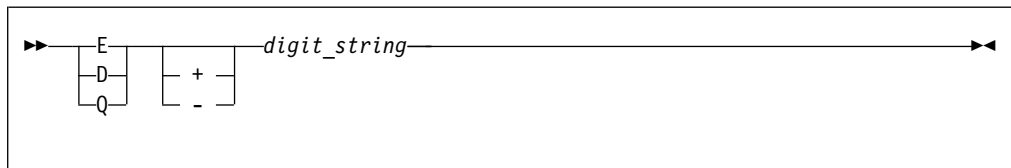
kind_param
> is either a *digit-string* or a *scalar-int-constant-name*

digit_string denotes a power of 10. **E** specifies a constant of type default real, unless you also include a *kind_param*, which overrides the default type. **D** specifies a constant of type default **DOUBLE PRECISION**. IBM **Q** specifies a constant of type **REAL(16)** in XL Fortran. IBM

If both *exponent* and *kind_param* are specified, the exponent letter must be **E**. If **D** or **Q** is specified, *kind_param* must not be specified.

A real literal constant that is specified without an exponent and a kind type parameter is of type default real.

Example of real constants

```
+0.
+5.432E02_16    !543.2 in 16-byte representation
7.E3
3.4Q-301
! Extended-precision constant
```

Complex

A complex type specifier must include one of the following keywords:
- The **COMPLEX** keyword.
- **IBM** The **DOUBLE COMPLEX** keyword. **IBM**

See "COMPLEX" on page 319 and "DOUBLE COMPLEX (IBM extension)" on page 341 for details on declaring entities of type complex.

─────────────────── **IBM extension** ───────────────────

The following table shows the corresponding values for the kind type parameter and the length specification when the complex type specifier has the **COMPLEX** keyword:

Kind Type Parameter *i* **COMPLEX(***i***)**	Length Specification *j* **COMPLEX****j*
4 8 16	8 16 32

In XL Fortran, the kind type parameter specifies the precision of each part of the complex entity, while the length specification specifies the length of the whole complex entity.

─────────────────── **End of IBM extension** ───────────────────

The kind of a complex constant is determined by the kind of the constants in the real and imaginary parts.

The precision of **DOUBLE COMPLEX** values is twice that of default complex values.

Scalar values of type complex can be formed using complex constructors. The form of a complex constructor is:

```
►►──(──expression──,──expression──)────────────────────────►◄
```

A complex literal constant is a complex constructor where each expression is a pair of constant expressions. Variables and expressions can be used in each part of the complex constructor as an XL Fortran extension.

In Fortran 95 you are only allowed to use a single signed integer, or real literal constant in each part of the complex constructor. In Fortran 2003, you can also use a named constant. F2008

In Fortran 2008, you can use complex part designators to access the real or imaginary part of complex entities directly. The type of a complex part designator is real and its kind and shape are those of the designator that appears to the left of the complex part selector. A complex part selector is either %RE or %IM. %RE selects the real part of a complex entity and %IM selects the imaginary part of a complex entity. Here is the syntax for complex part designators where *designator* has to be of type complex:

```
>>--designator--%----IM------------------------------><
                  |-RE-|
```
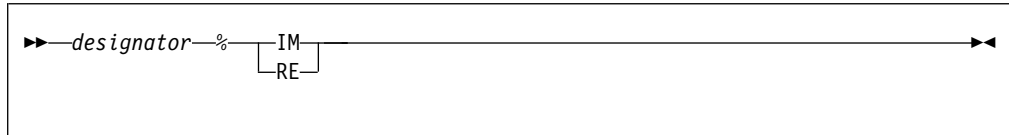
Complex part designators follow the rules for real data types. In addition, you can use complex part designators as variables in assignment statements; if x is of type complex, x%IM=0.0 sets the imaginary part of x to zero. F2008

If both parts of the literal constant are of type real, the kind type parameter of the literal constant is the kind parameter of the part with the greater precision, and the kind type parameter of the part with lower precision is converted to that of the other part.

If both parts are of type integer, they are each converted to type default real. If one part is of type integer and the other is of type real, the integer is converted to type real with the precision of the real part.

See "COMPLEX" on page 319 and "DOUBLE COMPLEX (IBM extension)" on page 341 for details on declaring entities of type complex.

Each part of a complex number has the same internal representation as a real number with the same kind type parameter.

Examples of complex constants

```
(3_2,-1.86)  ! Integer constant 3 is converted to default real
             ! for constant 3.0.
(45Q6,6D45)  ! The imaginary part is converted to extended
             ! precision 6.Q45.
(1+1,2+2)    ! Use of constant expressions. Both parts are
             ! converted to default real.
```

Examples of complex part designators

```
COMPLEX :: x, y, z
print *, x%RE ! Prints the same value as REAL(x)
print *, y%IM ! Prints the same value as AIMAG(y)
z%IM = 0.0    ! Sets the imaginary part of z to zero
```

Logical

The following table shows the values that XL Fortran can represent using the logical data type:

Kind parameter	Values	Internal (hex) Representation
1		
	.TRUE.	01
	.FALSE.	00
2		
	.TRUE.	0001
	.FALSE.	0000
4		
	.TRUE.	00000001
	.FALSE.	00000000
8		
	.TRUE.	0000000000000001
	.FALSE.	0000000000000000

Note: Any internal representation other than 1 for .TRUE. and 0 for .FALSE. is undefined.

XL Fortran sets the default kind type parameter to 4. The kind type parameter is equivalent to the byte size for logical values. Use the **-qintsize** compiler option to change the default logical size to 2, 4, or 8 bytes. Note that the **-qintsize** option similarly affects the default integer size. Use **–qintlog** to mix integer and logical data entities in expressions and statements.

The **-qport=clogicals** option allows you to instruct the compiler to treat all nonzero integers used in logical expressions as TRUE. In order to use the **-qport=clogicals** option, you must also specify the **-qintlog** option.

The logical type specifier must include the **LOGICAL** keyword. See "LOGICAL" on page 409 for details on declaring entities of type logical.

The form of a logical literal constant is:

```
►►──┬─.TRUE.──┬──────────────────────────────►◄
    └─.FALSE.─┘  └─_─kind_param─┘
```
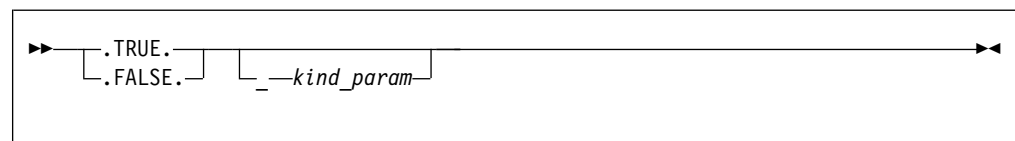
kind_param
    is either a *digit-string* or a *scalar-int-constant-name*

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

IBM You can also use the abbreviations T and F (without the periods) for .TRUE. and .FALSE., respectively, but only in formatted input, or as initial values in **DATA**

statements, **STATIC** statements, or type declaration statements. A kind type parameter cannot be specified for the abbreviated form. If T or F has been defined as a named constant, it is treated as that named constant rather than the logical literal constant. `IBM`

Example of a logical constant

```
.FALSE._4
 .TRUE.
```

Character

The character type specifier must include the **CHARACTER** keyword. See "CHARACTER" on page 307 for details on declaring entities of type character.

The form of a character literal constant is:
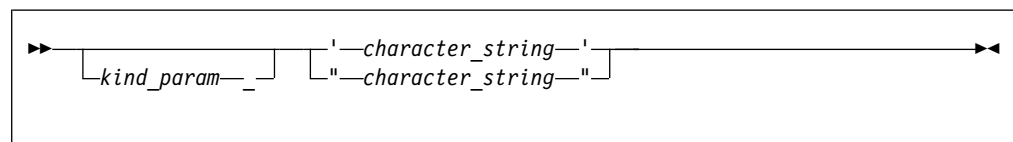
kind_param
> is either a *digit-string* or a *scalar-int-constant-name*

`IBM` XL Fortran supports a kind type parameter value of 1, representing the ASCII collating sequence. `IBM`

Character literal constants can be delimited by double quotation marks as well as apostrophes.

character_string consists of any characters capable of representation in XL Fortran, except the new-line character, because it is interpreted as the end of the source line. The delimiting apostrophes (') or double quotation marks (") are not part of the data represented by the constant. Blanks embedded between these delimiters are significant.

If a string is delimited by apostrophes, you can represent an apostrophe within the string with two consecutive apostrophes (without intervening blanks). If a string is delimited by double quotation marks, you can represent a double quotation mark within the string with two consecutive double quotation marks (without intervening blanks). The two consecutive apostrophes or double quotation marks will be treated as one character.

You can place a double quotation mark within a character literal 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 length of a character literal constant is the number of characters between the delimiters, except that each pair of consecutive apostrophes or double quotation marks counts as one character.

A zero-length character object uses no storage.

`IBM` In XL Fortran each character object requires 1 byte of storage.

For compatibility with C language usage, XL Fortran recognizes the following escape sequences in character strings:

Escape	Meaning
\b	Backspace
\f	Form feed
\n	New-line
\r	New-line
\t	Tab
\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

To ensure that scalar character constant expressions in procedure references are terminated with null characters (\0) for C compatibility, use the **-qnullterm** compiler option. (See **-qnullterm** options in the *XL Fortran Compiler Reference* for details and exceptions).

All escape sequences represent a single character. **IBM**

If you do not want these escape sequences treated as a single character, specify the **-qnoescape** compiler option. (See **-qescape** option in the *XL Fortran Compiler Reference*.) The backslash will have no special significance.

The maximum length of a character literal constant depends on the maximum number of characters allowed in a statement.

IBM If you specify the **-qctyplss** compiler option, character constant expressions are treated as if they are Hollerith constants. See "Hollerith constants" on page 32 for information on Hollerith constants. For information on the **-qctyplss** compiler option, see **-qctyplss** option in the *XL Fortran Compiler Reference*

XL Fortran supports multibyte characters within character literal constants, Hollerith constants, **H** edit descriptors, character-string edit descriptors, and comments through the **-qmbcs** compiler option.

Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames. See the *XL Fortran Compiler Reference* for more information. **IBM**

Examples of character constants

Example 1:
```
''                    ! Zero-length character constant.
```

Example 2:
```
1_"ABCDEFGHIJ"          ! Character constant of length 10, with kind 1.
```

IBM **Example 3:**
```
'\"\2\'\A567\\\\\''    ! Character constant of length 10 "2'A567\\'.
```

IBM

Character substrings

A character substring is a contiguous portion of a character string (called a parent string), which is a scalar variable name, scalar constant, scalar structure component, or array element. A character substring is identified by a substring reference whose form is:
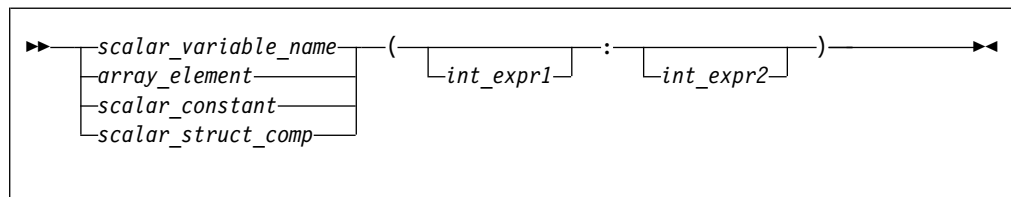
int_expr1 **and** *int_expr2*
> specify the leftmost character position and rightmost character position, respectively, of the substring. Each is a scalar integer expression called a substring expression.

The length of a character substring is the result of the evaluation of MAX(*int_expr2 - int_expr1 + 1*,0).

If *int_expr1* is less than or equal to *int_expr2*, their values must be such that:

- $1 \leq int\_expr1 \leq int\_expr2 \leq length$

where *length* is the length of the parent string. If *int_expr1* is omitted, its default value is 1. If *int_expr2* is omitted, its default value is *length*.

IBM FORTRAN 77 does not allow character substrings of length 0. Fortran 90 and up does allow these substrings. To perform compile-time checking on substring bounds in accordance with FORTRAN 77 rules, use the **-qnozerosize** compiler option. For Fortran 90 compliance, use **-qzerosize**. To perform run-time checking on substring bounds, use both the **-qcheck** option and the **-qzerosize** (or **-qnozerosize**) option. (See the *XL Fortran Compiler Reference* for more information.) **IBM**

A substring of an array section is treated differently. See "Substring ranges" on page 92.

Examples
```
CHARACTER(8) ABC, X, Y, Z
ABC = 'ABCDEFGHIJKL'(1:8)   ! Substring of a constant

X = ABC(3:5)                ! X = 'CDE'
Y = ABC(-1:6)               ! Not allowed in either FORTRAN 77 or Fortran 90

Z = ABC(6:-1)               ! Z = '' valid only in Fortran 90
```

Byte (IBM extension)

The byte type specifier is the **BYTE** keyword in XL Fortran. See "BYTE (IBM extension)" on page 300 for details on declaring entities of type byte.

The **BYTE** intrinsic data type does not have its own literal constant form. A **BYTE** data object is treated as an **INTEGER(1)**, **LOGICAL(1)**, or **CHARACTER(1)** data object, depending on how it is used.

Vector (IBM extension)

An entity you declare using the **VECTOR** keyword as part of a type declaration statement is of a vector type. An entity of a vector type has the same type as another entity if both entities are vectors that contain elements of the same type and kind. Otherwise, the two entities are of different types. You must not include vector objects in formatted I/O.

A vector can be any of the following types:
- A **PIXEL** vector
- An **INTEGER** vector with elements that have a kind type parameter of 1, 2, 4, or 8
- An **UNSIGNED** vector with elements that have a kind type parameter of 1, 2, 4, or 8
- A **REAL** vector with elements that have a kind parameter of 4 or 8

Note: A vector must only be declared if your **-qarch** option specifies an architecture that supports the VMX or VSX instruction set extensions. The **INTEGER(8)**, **UNSIGNED(8)**, and **REAL(8)** types require an architecture that supports the VSX instruction set extensions.

A vector is always a 16-byte entity.
- **INTEGER(1)** or **UNSIGNED(1)** vectors contains 16 elements.
- **INTEGER(2)**, **UNSIGNED(2)**, or **PIXEL** vectors contain 8 elements.
- **INTEGER(4)**, **UNSIGNED(4)**, or **REAL(4)** vectors contain 4 elements.
- **INTEGER(8)**, **UNSIGNED(8)**, or **REAL(8)** vectors contain 2 elements.

If kind type parameters are not specified for elements of **INTEGER**, or **UNSIGNED** vectors, the elements have the default integer kind. If kind type parameters are not specified for elements of **REAL** vectors, the elements have the default real kind.

Vectors must be aligned on a 16-byte boundary. XL Fortran automatically aligns vectors to 16 bytes, except in the following cases, where:
- The vector is a component of a sequence type or a record structure.
- The vector is a component of a derived type that has the **BIND** attribute and you compile with the **-qalign=bindc=packed** or **-qalign=bindc=bit_packed** options. This aligns the vector to a one-byte boundary.
- The vector is a member of a common block.
- The vector is storage-associated with a member of a common block that does not have a 16-byte boundary alignment.
- The vector is storage-associated with an array element that does not have a 16-byte boundary alignment.

Use the *Vector Interlanguage Interoperability* table to determine the corresponding XL C/C++ vector type when passing vectors between XL C/C++ and XL Fortran.

Table 12. Vector interlanguage interoperability

XL Fortran vector type	XL C/C++ vector type
VECTOR(INTEGER(1))	vector signed char
VECTOR(INTEGER(2))	vector signed short
VECTOR(INTEGER(4))	vector signed int, vector signed long
VECTOR(INTEGER(8))	vector signed long long
VECTOR(PIXEL)	vector pixel
VECTOR(REAL(4))	vector float
VECTOR(REAL(8))	vector double
VECTOR(UNSIGNED(1))	vector unsigned char
VECTOR(UNSIGNED(2))	vector unsigned short
VECTOR(UNSIGNED(4))	vector unsigned int, vector unsigned long
VECTOR(UNSIGNED(8))	vector unsigned long long

Pixel (IBM extension)

The **PIXEL** keyword specifies the pixel type. A pixel is a two-byte entity that the compiler interprets in four parts. The first part consists of one bit. The remaining three parts consist of 5 bits each. Pixel literals are not supported. You must specify a pixel only as part of a vector declaration.

Unsigned (IBM extension)

The **UNSIGNED** keyword specifies the unsigned integer type. Use the **-qintsize** compiler option to change the default integer size to 2 or 4 bytes. The default kind type parameter is 4. Unsigned integer literals are not supported. You must specify the unsigned integer type only as part of a vector declaration.

Chapter 4. Derived types

A derived type is a composite data type that can contain both intrinsic and derived data types. You can define a derived type by using a type definition. This definition specifies the name of the derived type and its type parameters, components, and procedures. In Fortran 95, a type definition must have at least one component and must not contain procedures. In Fortran 2003, a type definition can have zero or more components, procedures and type parameters. Within a derived type, the names of type parameters, components and procedures must be unique, although the names can be the same as the names outside the scope of the derived type definition.

In Fortran 2003, a derived type can be parameterized by type parameters. Each type parameter is defined to be either a kind or a length type parameter, and can have a default value. For details, see "Derived type parameters (Fortran 2003)" on page 50.

Syntax of a derived type

```
►►─┬─DERIVED_TYPE_statement──────────────────────────────────►
                          └─type_param_def_block─┘

►─┬────────────────────────────┬─┬─────────────────────────┬─►
  ├─private_components_stmt─┘   └─component_def_stmt_block─┘
  └─SEQUENCE──────────────┘

►─┬──────────────────────────┬─END_TYPE_statement───────────►◄
  └─type_bound_procedure_block─┘
```

where *private_components_stmt* is as follows:

```
►►──PRIVATE─────────────────────────────────────────────────►◄
```
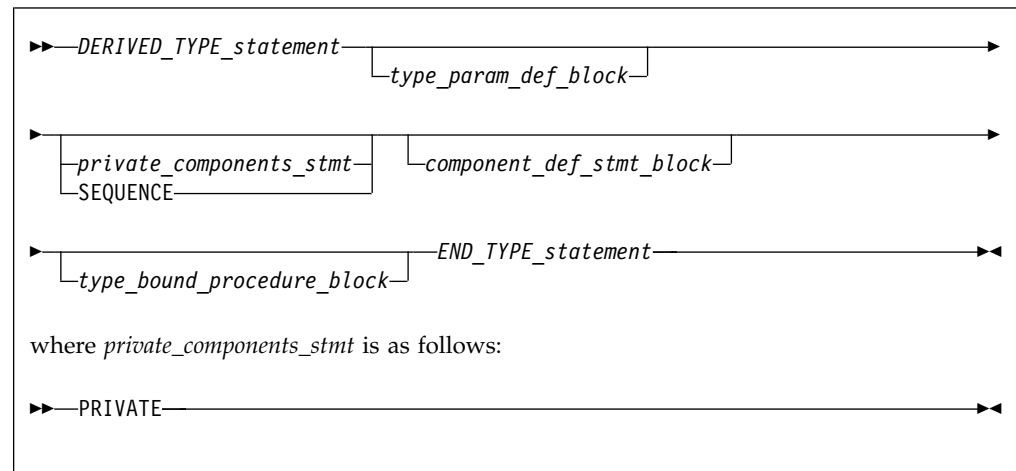
DERIVED_TYPE_statement
> See "TYPE" on page 333 for syntax details.

F2003 *type_param_def_block*
> Consists of the declarations for all the *type_param_names* that exist in *DERIVED_TYPE_statement*. For details, see Derived type parameters.

private_components_stmt
> Specifies that default accessibility for the components of the derived type are private. You can only specify one *private_components_stmt* for a given derived type. You can only specify a *private_components_stmt* if the type definition is within the specification part of a module.

SEQUENCE
> You can only specify one **SEQUENCE** statement. For details see "SEQUENCE" on page 460.

component_def_stmt_block
> A *component_def_stmt_block* consists of one or more type declaration

statements or procedure component declaration statements to define the components of the derived type. The attributes that a type declaration statement in *component_def_stmt_block* can specify include `F2003` ALLOCATABLE `F2003` , `F2008` CONTIGUOUS `F2008` , "DIMENSION" on page 335, POINTER, "PRIVATE" on page 432 and "PUBLIC" on page 440 attributes. For more information, see "Type Declaration" on page 476 and "Procedure pointer components" on page 53.

Fortran 2003
---

type_bound_procedure_block
> Consists of a **"CONTAINS" on page 322** statement, followed optionally by a *BINDING_PRIVATE_stmt* statement, and one or more procedure binding statements. *BINDING_PRIVATE_stmt* specifies that the default accessibility for the derived type bindings are private, and you can only specify one *BINDING_PRIVATE_stmt* binding statement for a given derived type. For details about *BINDING_PRIVATE_stmt*, see "Syntax of a type-bound procedure" on page 61.

End of Fortran 2003

END_TYPE_statement
> Optionally contains the same *type_name* as specified on the **TYPE** statement. For more information see "END TYPE" on page 355.

Derived type parameters (Fortran 2003)

A derived type is parameterized if the **DERIVED_TYPE_statement** has any **type_param_names**, or if it inherits any type parameter from its ancestor type. You can define the type parameters for the derived type.

Syntax of a type parameter definition

```
>>--INTEGER-------------------,--+-KIND-+--::--type_param_decl_list------><
                 |-kind_selector-|    +-LEN--+
```
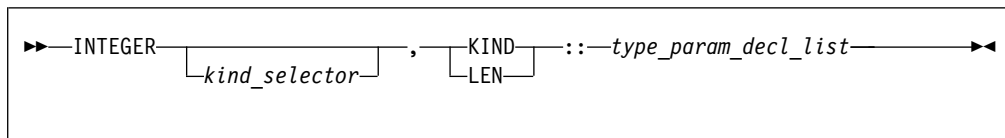
kind_selector
> Specifies the kind type parameter for the integer type. See the *kind_selector* section of INTEGER statement for syntax details.

type_param_decl

```
>>--type_param_name-----------------------------------------------------><
                   |-=--scalar_int_initialization_expr-|
```
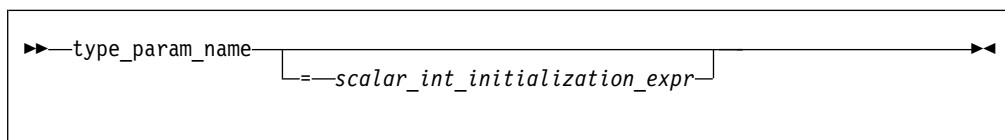
Each **type_param_name** in a **type_param_decl** must match one of the **type_param_name** parameters listed in the **DERIVED_TYPE_statement**. Each **type_param_name** in the **DERIVED_TYPE_statement** must be specified once in a **type_param_decl** in the derived type definition.

A derived type parameter can be one of the following parameters.
- It is a kind type parameter if it is declared with the **KIND** specifier.
- It is a length type parameter if it is declared with the **LEN** specifier.

A derived type parameter can be used as a primary in a specification expression in the derived type definition. A kind type parameter can also be used as a primary in a constant expression in the derived type definition.

If a **type_param_decl** has a **scalar_int_initialization_expr**, the type parameter has a default value specified by the expression.

Example

```
TYPE general_point (k, dim)
  INTEGER, KIND :: k = selected_real_kind(6,70)
  INTEGER, LEN :: dim

  REAL(k) :: coordinates(dim)
END TYPE general_point
```

Type parameter order

Is the sequence of type parameters of a parameterized derived type. It is used for derived type specifiers (see Type Declaration) that do not use type parameter name keywords.

If a type is not an extended type, its type parameter order is the order of the type parameter list in the **DERIVED_TYPE_statement**. The type parameter order of an extended type consists of the type parameter order of its parent type, followed by any additional type parameters in the order of the type parameter list in its **DERIVED_TYPE_statement**.

Derived type components

The components of a derived type can be of any intrinsic type or can be of a previously defined derived type. The components of a derived type can be either direct or ultimate.

Direct components of a derived type are components that satisfy any of the following conditions:
- The components of that derived type.
- The direct components of a derived type component that has neither the **ALLOCATABLE** nor the **POINTER** attribute.

Ultimate components are components that satisfy any of the following three conditions:
- Components of intrinsic data type.
- Components with the F2003 **ALLOCATABLE** F2003 or **POINTER** attribute.
- The ultimate components of a derived type component that has neither the F2003 **ALLOCATABLE** F2003 nor **POINTER** attribute.

You can specify a default initialization for each nonallocatable component in the definition of a derived type. Allocatable components are always initialized to deallocated.

An attribute cannot appear more than once in a type declaration statement or a procedure component declaration statement.

If you specify neither the `F2003` **ALLOCATABLE** `F2003` nor the **POINTER** attribute for a component, that component must be of an intrinsic type or a previously defined derived type.

If you specify neither the `F2003` **ALLOCATABLE** `F2003` nor the **POINTER** attribute for an array component, the array bounds must be explicitly specified for each dimension. Each bound must be an expression that does not include any references to specification functions or the following intrinsic functions: **ALLOCATED**, **ASSOCIATED**, **EX-TENDS TYPE OF**, **PRESENT**, or **SAME TYPE AS**. Each specification inquiry reference is a constant expression whose value does not depend on a variable value.

When you define a component, the value of a type parameter must be a colon or an expression that does not include any references to specification functions or the following intrinsic functions: **ALLOCATED**, **ASSO-CIATED**, **EXTENDS TYPE OF**, **PRESENT**, or **SAME TYPE AS**. Each specification inquiry reference is a constant expression whose value does not depend on a variable value.

A component of a derived type must not appear as an input/output list item if any ultimate component of the object cannot be accessed by the scoping unit of the input/output statement, `F2003` unless a user-defined input/output procedure processes the derived-type object. `F2003` A derived-type object must not appear in a data transfer statement if the object has a component that is a pointer or allocatable, `F2003` unless a user-defined input/output procedure processes the object. `F2003`

Allocatable and pointer components

A component is allocatable if it has the **ALLOCATABLE** attribute. A component is a pointer if it has the **POINTER** attribute. A component cannot have both the **ALLOCATABLE** and **POINTER** attributes.

A pointer component can be a data pointer or a procedure pointer. The type of a data pointer component can be the same as the type containing the component.

Allocatable components are defined as ultimate components just as pointer components are. This is because the value (if any) is stored separately from the rest of the structure, and this storage does not exist (because the object is unallocated) when the structure is created. As with pointer components, variables containing allocatable components are forbidden from appearing directly in input/output lists, unless the variable is processed by a user-defined derived type input/output procedure.

As with allocatable arrays, allocatable components are forbidden from storage association contexts. So, any variable containing a pointer or allocatable component cannot appear in **COMMON** or **EQUIVALENCE**. However, allocatable components are permitted in **SEQUENCE** types, which allows the same type to be defined separately in more than one scoping unit.

If a component of type character or derived type has the **ALLOCATABLE** or **POINTER** attribute, it can have deferred length.

Examples

```
MODULE REAL_POLYNOMIAL_MODULE
  TYPE REAL_POLYNOMIAL
    REAL, ALLOCATABLE :: COEFF(:)
  END TYPE
```

```
      INTERFACE OPERATOR(+)
        MODULE PROCEDURE RP_ADD_RP, RP_ADD_R
      END INTERFACE
CONTAINS
      FUNCTION RP_ADD_R(P1,R)
        TYPE(REAL_POLYNOMIAL) RP_ADD_R, P1
        REAL R
        INTENT(IN) P1,R
        ALLOCATE(RP_ADD_R%COEFF(SIZE(P1%COEFF)))
        RP_ADD_R%COEFF = P1%COEFF
        RP_ADD_R%COEFF(1) = P1%COEFF(1) + R
      END FUNCTION
      FUNCTION RP_ADD_RP(P1,P2)
        TYPE(REAL_POLYNOMIAL) RP_ADD_RP, P1, P2
        INTENT(IN) P1, P2
        INTEGER M
        ALLOCATE(RP_ADD_RP%COEFF(MAX(SIZE(P1%COEFF), SIZE(P2%COEFF))))
        M = MIN(SIZE(P1%COEFF), SIZE(P2%COEFF))
        RP_ADD_RP%COEFF(:M) = P1%COEFF(:M) + P2%COEFF(:M)
        IF (SIZE(P1%COEFF)>M) THEN
          RP_ADD_RP%COEFF(M+1:) = P1%COEFF(M+1:)
        ELSE IF (SIZE(P2%COEFF)>M) THEN
          RP_ADD_RP%COEFF(M+1:) = P2%COEFF(M+1:)
        END IF
      END FUNCTION
END MODULE

PROGRAM EXAMPLE
  USE REAL_POLYNOMIAL_MODULE
  TYPE(REAL_POLYNOMIAL) P, Q, R
  P = REAL_POLYNOMIAL((/4,2,1/)) ! Set P to (X**2+2X+4)
  Q = REAL_POLYNOMIAL((/1,1/)) ! Set Q to (X+1)
  R = P + Q ! Polynomial addition
  PRINT *, 'Coefficients are: ', R%COEFF
END
```

Related information

- "Procedure pointer components"

Procedure pointer components

Syntax of a procedure pointer component
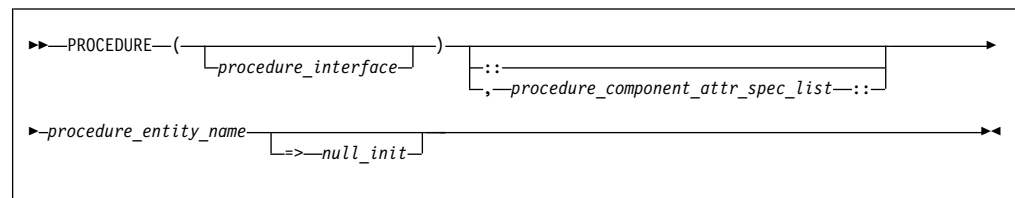
procedure_interface

> A *declaration_type_spec* or the name of a procedure that has an explicit interface. For more information, see "Type Declaration" on page 476.
>
> If *procedure_interface* is a *declaration_type_spec* and the type specified is a parameterized derived type, all parameters used must be known at compile time.

procedure_component_attr_spec_list
> Attributes from the following list:
>
> - **POINTER**
> - **PASS**

- **NOPASS**
- *access_spec*

procedure_entity_name
> is the name of the procedure pointer that is being declared.

null_init
> is a reference to the NULL intrinsic function.

The **PASS** attribute defines the passed-object dummy arguments of the type-bound procedure or procedure pointer component. When **NOPASS** is specified the procedure has no passed-object dummy argument.

PASS or **NOPASS** shall not both appear in the same *procedure_component_attr_spec_list*.

If the procedure pointer component has an implicit interface or has no arguments, **NOPASS** shall be specified.

If **PASS**(*arg-name*) appears, the interface shall have a dummy argument named *arg-name* which is the passed-object dummy arguments.

The passed-object dummy arguments must not be a pointer, must not be allocatable, and all its length type parameters must be assumed.

If neither **PASS** nor **NOPASS** is specified or **PASS** has no *arg-name*, the first dummy argument is the passed-object dummy argument.

POINTER must be present in each *procedure_component_attr_spec_list*.

Related information
- "Procedure pointers (Fortran 2003)" on page 185
- "Passed-object dummy arguments" on page 63

Array components

A derived type component can be an array. For details, see "Array sections" on page 88, "Array sections and structure components" on page 92, "Array declarators" on page 76, and "Type Declaration" on page 476.

Nonpointer, nonallocatable array components can be declared with either constant dimension declarators or specification expressions that can involve type parameters.

Note: You must declare pointer and allocatable array components with a *deferred_shape_spec_list* array specification. For details, see "Deferred-shape arrays" on page 80.

F2008 If the **CONTIGUOUS** attribute is specified, the component must be an array with the **POINTER** attribute. F2008

Default initialization for components

You cannot specify default initialization for an allocatable component. Allocatable components are always initialized to unallocated.

You can specify default initialization for a nonpointer component using an equal sign followed by a constant expression, or by enclosing an *initial_value_list* in slashes. Enclosing an *initial_value_list* in slashes can apply to components in a standard derived type declaration, or those within a record structure.

For pointer components default initialization, use an arrow (=>) and then a reference to the **NULL** intrinsic with no arguments. You can specify default initialization for zero or more components of a derived type, but it is not necessary for every component.

A data object has default initialization if any direct component of its type is initialized. **IBM** Such an object cannot be a pointee. For more information about pointees, see "POINTER (integer) (IBM extension)" on page 428.**IBM**

If a nonpointer component is of a type that has default initialization, the default initialization specified for the component overrides the default initialization specified for the components of the type.

IBM You can use an object that has default initialization in a common block as an IBM extension. **IBM**

Unlike explicit initialization, the **SAVE** attribute is not implied by default initialization. **IBM** The **-qsave=defaultinit** option causes default initialization to imply the **SAVE** attribute.**IBM**

If a storage unit that is storage associated has default initialization, the objects or subobjects supplying the default initialization must be of the same type and type parameters, bounds, and supply the same value for that storage unit.

A direct component receives an initial value if you specify default initialization on the corresponding component definition in the type definition, regardless of the accessibility of that component.

For a data object that can undergo default initialization, its nonpointer components are either initially undefined, or their corresponding default initialization expressions define them. Its pointer components with default initialization have association status disassociated, and all other pointer components have association status undefined.

If a variable that has default initialization has static storage class, then default initialization occurs for that variable when your application executes.

If a variable that is a function result, an **INTENT(OUT)** dummy argument, or a local variable without the **SAVE** attribute has default initialization, then default initialization occurs when the procedure containing the variable's declaration executes.

Allocation of an object that has default initialization for a component causes the component to:
- Become defined, if the component is a nonpointer
- Become disassociated, if the component is a pointer.

In a subprogram with an **ENTRY** statement, default initialization occurs only for the dummy arguments that appear in the argument list of the procedure name the

ENTRY statement references. If a dummy argument has the **OPTIONAL** attribute, default initialization occurs only if that dummy argument is present.

Module data objects of derived type with default initialization must have the **SAVE** attribute to be a candidate for default initialization.

Component order

Component order is the sequence of non-parent components of a derived type as the sequence pertains to list-directed and namelist formatted input/output, and structure constructors that do not use component keywords.

If a type is not an extended type, its component order is the order of the declarations of the components in the derived type definition. The component order of an F2003 extended type F2003 consists of the component order of its parent type, followed by any additional components in the order of their declarations in the extended derived type definition.

Referencing components

You can refer to a specific structure component using a *component designator*. A scalar component designator has the following syntax:

scalar_struct_comp:
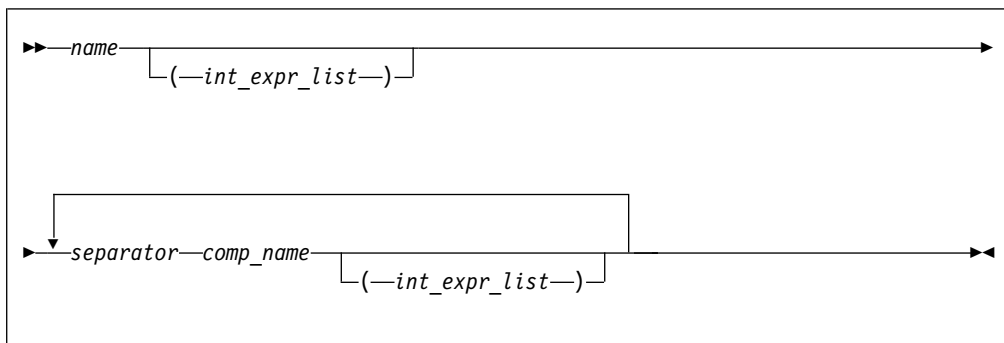
name   is the name of an object of derived type

comp_name
    is the name of a derived type component

int_expr
    is a scalar integer or real expression called a subscript expression

separator
    is % or IBM . IBM

The structure component has the same type, type parameters, and **POINTER** attribute (if any) as the right-most *comp_name*. It inherits any **INTENT**, **TARGET**, and **PARAMETER** attributes from the parent object.

Note:
- Each *comp_name* must be a component of the declared type of the immediate preceding *name* or *comp_name*.
- The *name* and each *comp_name*, except the right-most, must be of derived type.
- The number of subscript expressions in any *int_expr_list* must equal the rank of the preceding *name* or *comp_name*.

- The *name* must be the name of a data object.
- Only *name* or at most one *comp_name* can have nonzero rank. If *name* or a *comp_name* has nonzero rank, the *comp_name* to the right must not have the **ALLOCATABLE** or **POINTER** attribute.
- The rank of the structure component is the rank of the name of *comp_name* that has nonzero rank (if any). Otherwise, the rank is zero.
- ▸F2003◂ If the declared type of the rightmost *comp_name* is of abstract type, the structure component must be polymorphic. ◂F2003◂

In namelist formatting, a separator must be a percentage sign.

If an expression has a form that could be interpreted either as a structure component using periods as separators or as a binary operation, and an operator with that name is accessible in the scoping unit, XL Fortran will treat the expression as a binary operation. If that is not the interpretation you intended, you should use the percent sign to dereference the parts, or, in free source form, insert white space between the periods and the *comp_name*.

Examples

Example 1: Ambiguous use of a period as separator

```
MODULE MOD
  STRUCTURE /S1/
    STRUCTURE /S2/ BLUE
      INTEGER I
    END STRUCTURE
  END STRUCTURE
  INTERFACE OPERATOR(.BLUE.)
    MODULE PROCEDURE BLUE
  END INTERFACE
CONTAINS
  INTEGER FUNCTION BLUE(R1, I)
    RECORD /S1/ R1
    INTENT(IN) :: R1
    INTEGER, INTENT(IN) :: I
    BLUE = R1%BLUE%I + I
  END FUNCTION BLUE
END MODULE MOD

PROGRAM P
  USE MOD
  RECORD /S1/ R1
  R1%BLUE%I = 17
  I = 13
  PRINT *, R1.BLUE.I ! Calls BLUE(R1,I) - prints 30
  PRINT *, R1%BLUE%I ! Prints 17
END PROGRAM P
```

Example 2: Mix of separators

```
STRUCTURE /S1/
  INTEGER I
END STRUCTURE
STRUCTURE /S2/
  RECORD /S1/ C
END STRUCTURE
RECORD /S2/ R
R.C%I = 17 ! OK
R%C.I = 3 ! OK
R.C.I = 19 ! OK
END
```

Example 3: Percent and period work for any derived types
```
      STRUCTURE /S/
        INTEGER I, J
      END STRUCTURE
      TYPE DT
        INTEGER I, J
      END TYPE DT
      RECORD /S/ R1
      TYPE(DT) :: R2
      R1.I = 17; R1%J = 13
      R2.I = 19; R2%J = 11
      END
```

Component and procedure accessibility

The default accessibility of a component of a derived type is **PUBLIC**. The **PRIVATE** components statement changes that default accessibility to private.

You can use the **PRIVATE** or **PUBLIC** attribute on each component of a derived type to override the default accessibility. If a component is private, the component name is accessible only within the module containing the derived type definition, even if the derived type itself is public.

Fortran 2003

The default accessibility of a procedure binding is **PUBLIC**. The *BINDING_PRIVATE_stmt* changes that default accessibility to private. You can use the **PRIVATE** or **PUBLIC** attribute on a procedure binding to override the default accessibility. If the procedure binding is private, it is accessible only within the defining module, even if the derived type itself is public.

End of Fortran 2003

Sequence derived types

By default, the order of derived type component definitions does not imply a storage sequence. However, if you include a **SEQUENCE** statement, the derived type becomes a sequence derived type. For a sequence derived type, the order of the components specifies a storage sequence for objects of this derived type. If a component of a sequence derived type is of a derived type, that derived type must also be a sequence derived type.

Attention:

Using sequence derived types can lead to misaligned data, which can adversely affect the performance of your application. Use with discretion.

Extensible derived types (Fortran 2003)

An extensible type is a nonsequence noninteroperable derived type from which you can extend new types. You cannot use record structure syntax to define an extensible type. You can further classify an extensible type to be one or more of the following:

Base type
Extends only itself and no other types.

Extended type

Extends not only itself, but all types for which its parent type is an extension.

Parent type

Provides components and procedure bindings to all types that extend from that type. A parent type is the extensible type from which an extended type is derived

You define an extended type with the **EXTENDS** attribute. The **EXTENDS** attribute specification includes the name of the parent type. For more information on specifying the **EXTENDS** attribute see "TYPE" on page 333.

An extended type inherits all of the type parameters, components and nonoverridden, nonfinal procedure bindings from its parent type.

The extended type also inherits inaccessible components and bindings from the parent type. They remain inaccessible in the extended type. A private entity is inaccessible if the type that you extend is accessed through use association.

A base type is not required to have any type parameters, components or bindings. An extended type is not required to have more type parameters, components or bindings than its parent type.

A type is not required to use any type parameters it or any parent may have defined.

An extended type has a scalar, nonpointer, nonallocatable, parent component with the same type and type parameters as its parent type. The name of this component is identical to the name of the parent type, and has the same accessibility.

A type parameter or component declared in an extended type must not have the same name as any type parameter or component of its parent type.

Example of an extended type

```
TYPE :: POINT ! A base type
   REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
   INTEGER :: COLOR                 ! Components X and Y, and component name
END TYPE COLOR_POINT                ! POINT, inherited from parent
```

In the example, the type COLOR_POINT inherits the components X and Y from parent type POINT. The components retain all of the attributes they have in the parent type. You can declare additional components and procedure bindings in the derived type definition of the extended type. In the example of an extensible type, the type COLOR_POINT has one additional component, COLOR. The type POINT is a nonsequence type that is not an extension of another type and therefore a base type. The type COLOR_POINT is an extended type, whose parent type is POINT.

COLOR_POINT has a parent component POINT. The parent component, POINT, is a structure with the components X and Y. Components of the parent are inheritance associated with the corresponding components inherited from the parent type. An ancestor component of a type is the parent component of the type or an ancestor component of the parent component. The ancestor component of COLOR_POINT is the parent component POINT.

For code example of type parameters, see the "Type Declaration" on page 476 section.

Abstract types and deferred bindings (Fortran 2003)

An abstract type is a type with the **ABSTRACT** attribute. A nonpolymorphic object must not be declared with an abstract type.

A binding with the **DEFERRED** attribute is a deferred binding. A deferred binding defers the implementation of a procedure to extensions of the type. You can specify a deferred binding only in an abstract type definition. The dynamic type of a polymorphic object cannot be an abstract type. Therefore, a deferred binding cannot be invoked. An extension of an abstract type does not have to be abstract if that extension does not contain or inherit deferred bindings.

If a type definition contains or inherits a deferred binding, the **ABSTRACT** attribute must appear. If **ABSTRACT** appears, the type must be extensible.

Example of an abstract type

```
TYPE, ABSTRACT :: FILE_HANDLE
   CONTAINS
   PROCEDURE(OPEN_FILE), DEFERRED, PASS(HANDLE) :: OPEN
   ...
END TYPE

INTERFACE
SUBROUTINE OPEN_FILE(HANDLE)
IMPORT FILE_HANDLE
   CLASS(FILE_HANDLE), INTENT(IN):: HANDLE
END SUBROUTINE OPEN_FILE
END INTERFACE
```

Derived type values

The set of values of a particular derived type consists of all possible sequences of the component values of its components. The following table lists component values of different types of components.

Table 13. Component values

Component	Component value
Associated pointer	Allocation status, dynamic type and type parameters, bounds and value
Disassociated pointer	Association status
Unallocated allocatable	Allocation status
Allocated allocatable	Allocation status, dynamic type and type parameters, bounds and value
Nonpointer nonallocatable	Value

Type-bound procedures (Fortran 2003)

The procedure portion of a derived type definition allows you to bind specific procedures, generic interfaces, and final subroutines to a type.

Syntax of a type-bound procedure

The syntax of the type-bound procedure portion of a derived type definition is as follows:
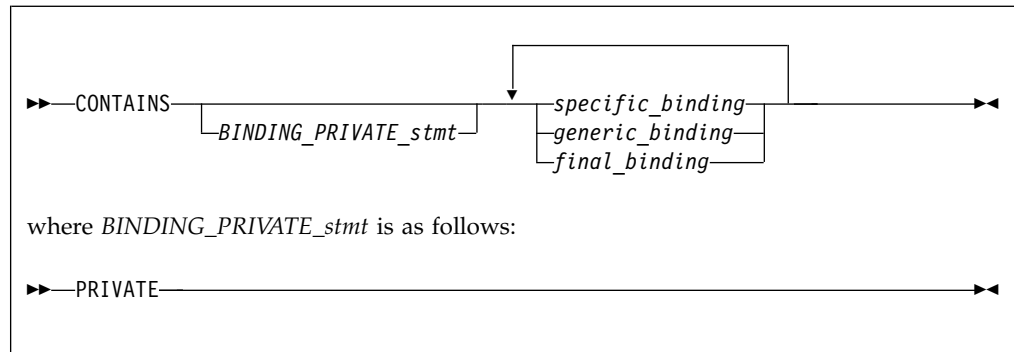
```
>>─CONTAINS──────────┬──────────────────────────────────┬──><
                     └─BINDING_PRIVATE_stmt─┘   ┌─◄─────────┐
                                              ▼ ┌─specific_binding─┐
                                                ├─generic_binding──┤
                                                └─final_binding────┘

where BINDING_PRIVATE_stmt is as follows:

>>─PRIVATE──────────────────────────────────────────────────><
```

CONTAINS
>   For more information see "CONTAINS" on page 322

BINDING_PRIVATE_stmt
>   You can only specify a *BINDING_PRIVATE_stmt* if the type definition is within the specification part of a module.

specific_binding
>   Binds a procedure to the type, or specifies a deferred binding in an abstract type. See "Specific binding"

generic_binding
>   Defines a generic interface. See "Generic binding" on page 63

final_binding
>   Defines a list of final subroutines. See "Final binding" on page 65

You can identify a procedure using a binding name in the scope of the type definition, or an operator for a generic binding. The binding name is the name of a procedure of the type name and is referred to in the same way as a component of a type. For a specific binding, this name is the *binding_name*. For a generic binding whose generic specification is *generic_name*, this name is the *generic_name*. A final binding, or a generic binding whose generic specification is not *generic_name*, has no binding name.

Specific binding

Syntax of a *specific_binding*

The form of specific binding is:

```
►►──PROCEDURE─────────────────────────────────────────────────────────────►
               └─(──interface_name──)─┘  ├─,──attribute_list──::─┤
                                         └─::──────────────────────┘


         ┌─────────,─────────┐
         │         (1)       │
   ►─────▼──binding_name──────┴────────────────────────────────────►◄
              └─=>──procedure_name─┘

Notes:

1     Fortran 2008
```
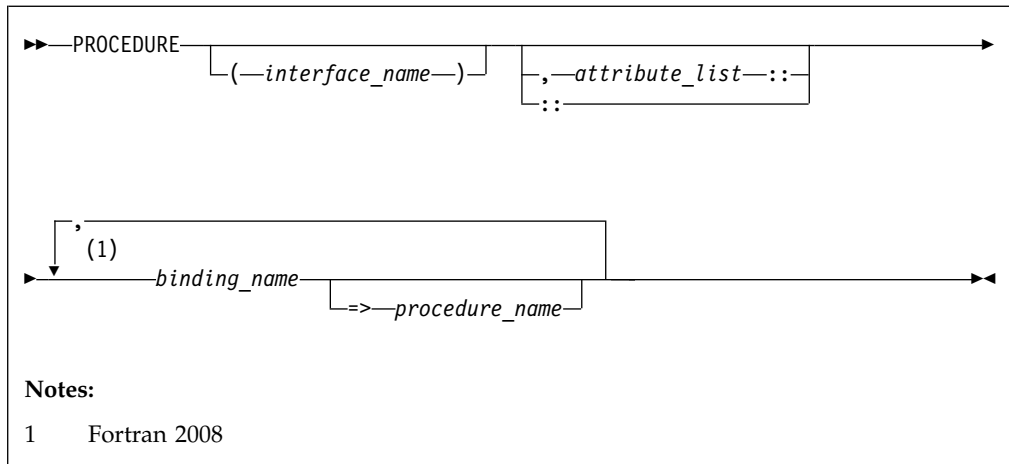
interface_name

> defines the interface of the type-bound procedure. The *interface_name* must be the name of an abstract interface or of a procedure that has an explicit interface. If you specify an *interface_name*, you must not specify a *procedure_name*. An interface-name can appear if and only if the binding has the **DEFERRED** attribute.

attribute

> A binding can have one or more attributes, called binding attributes. The same binding attribute cannot appear more than once for the same binding. The list of binding attributes that you specify in an *attribute_list* includes:

> **PASS**   Defines the passed-object dummy argument of the procedure.

> **NOPASS**
> > Indicates that the procedure has no passed-object dummy argument. If the interface of the binding has no dummy argument of the type being defined, use **NOPASS**. **PASS** and **NOPASS** can not both be specified for the same binding.

> *access_spec*
> > Is **PUBLIC** or **PRIVATE**.

> **NON_OVERRIDABLE**
> > Prevents a binding from being overridden in an extended type. You must not specify **NON_OVERRIDABLE** for a binding with the **DEFERRED** attribute.

> **DEFERRED**
> > Marks the procedure as deferred. Deferred bindings must only be specified for derived type definitions with the **ABSTRACT** attribute. A procedure with the **DEFERRED** binding attribute must specify an *interface_name*. An overriding binding can have the **DEFERRED** attribute only if the binding it overrides is deferred. The **NON_OVERRIDABLE** and **DEFERRED** binding attributes must not both be specified for the same procedure. See "Abstract types and deferred bindings (Fortran 2003)" on page 60 and "Procedure overriding" on page 67 for more information.

binding_name

> is the name of a binding of a type.

procedure_name
>   defines the interface of the type-bound procedure. The *procedure_name* must
>   be the name of an accessible module procedure or an external procedure
>   that has an explicit interface. If neither =>*procedure_name* nor *interface_name*
>   appears, the *procedure_name* is the same as the *binding_name*. If
>   =>*procedure_name* appears, you must specify the double-colon separator
>   and an *interface_name* must not be specified.

Passed-object dummy arguments

A passed-object dummy argument applies to a type-bound procedure, or a
procedure pointer component.

- If you specify **PASS** (*arg-name*) the interface of the procedure pointer component
  or named type-bound procedure has a dummy argument with the same name as
  arg-name. In this case, the passed-object dummy argument is the argument with
  the given name.

- If you do not specify **PASS** or **NOPASS**, or specify **PASS** without *arg-name*, the
  first dummy argument of a procedure pointer component or type-bound
  procedure is the passed-object dummy argument.

The passed-object dummy argument must be a scalar, nonpointer, nonallocatable
dummy data object with the same declared type as the type being defined. All of
its length type parameters must be assumed. The dummy argument must be
polymorphic if and only if the type being defined is extensible.

In the example of a type-bound procedure with a specific binding, the type POINT
contains a type-bound procedure with a specific binding. LENGTH is the type-bound
procedure and POINT_LENGTH is the name of a module procedure.

Example of a type-bound procedure with a specific binding

```
MODULE smaple
TYPE :: POINT
   REAL :: X, Y
   CONTAINS
      PROCEDURE, PASS :: LENGTH => POINT_LENGTH
END TYPE

CONTAINS
   REAL FUNCTION point_length (a, b)

      CLASS (POINT), INTENT (IN) :: a, b
      point_length = SQRT ( (a%X - b%X)**2 + (a%Y - b%Y)**2 )

   END FUNCTION point_length
END MODULE
```

Generic binding

Syntax of a *generic_binding*

The form of generic_binding is:

```
►►──GENERIC─────────────────────┬───::─generic_spec─=>─binding_name_list────────────►◄
              ├─,─PRIVATE─┤
              └─,─PUBLIC──┘
```
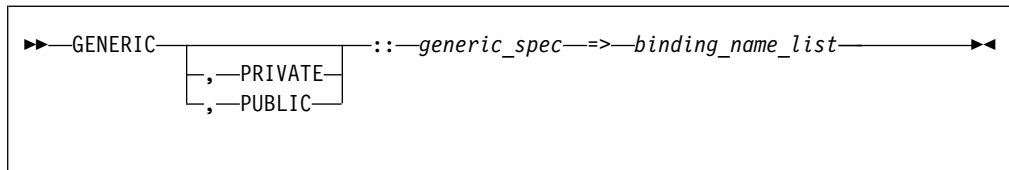
The *generic_spec* can be any of the following:

generic_name

OPERATOR(*defined-operator***)**
> The interface of each binding must be as specified in "Defined operators" on page 170.

ASSIGNMENT(=)
> The interface of each binding must be as specified in "Defined assignment" on page 172.

dtio_generic_spec
> The interface of each binding must be as specified in "User-defined derived-type Input/Output procedure interfaces (Fortran 2003)" on page 222.

If the *generic_spec* is a *generic_name*, the *generic_name* cannot be the name of a nongeneric binding of the type. The same *generic_spec* may be used in several generic bindings within a single derived-type definition. In this case, every occurrence of the same *generic_spec* must have the same accessibility. Each binding name in the *binding_name_list* must be the name of a specific binding of the type.

When *generic_spec* is not a *generic_name*, each specific binding name in the *binding_name_list* must have the passed-object dummy argument. You can only specify one binding attribute, **PRIVATE** or **PUBLIC**. The following is an example of a generic binding where *generic_spec* is **ASSIGNMENT(=)**.

Examples
```
! See example of a procedure with a specific binding for definition of COLOR_POINT
TYPE, EXTENDS(color_point) :: point_info
! An extension of TYPE(COLOR_POINT)

   REAL :: color_code
   CONTAINS
   PROCEDURE, NOPASS:: get_color_code
   PROCEDURE :: info1 => color_to_info
   PROCEDURE :: point1 => point_to_info
   GENERIC :: ASSIGNMENT(=) => info1, point1
END TYPE point_info

CONTAINS
ELEMENTAL SUBROUTINE color_to_info(a, b)
   CLASS(point_info), INTENT(OUT) :: a
   TYPE(color_point), INTENT(IN):: b
   a%color_point = b
   a%color_code = get_color_code(b%color)
END SUBROUTINE
ELEMENTAL SUBROUTINE point_to_info(a, b)
   CLASS(point_info), INTENT(OUT) :: a
   TYPE(point), INTENT(IN):: b
   a%color_point = color_point(point=b, color=1)
   a%color_code = get_color_code(1)
END SUBROUTINE
```

The following is an example of type parameters that illustrates how length parameters should be used. As illustrated in the example, multiple procedures must be defined for multiple kind parameter values:

```
! Separate specific bindings may be needed for multiple possible kind parameters:
TYPE :: GRAPH (PREC,NNODES)
  INTEGER, KIND :: PREC
  INTEGER, LEN  :: NNODES
  REAL(PREC) :: XVAL(NNODES), YVAL(NNODES)
CONTAINS
  PROCEDURE, PASS :: FINDMAX_Y_4
  PROCEDURE, PASS :: FINDMAX_Y_8
  GENERIC :: FINDMAX_Y => FINDMAX_Y_4, FINDMAX_Y_8
END TYPE GRAPH

CONTAINS
INTEGER FUNCTION FINDMAX_Y_4(G)
  CLASS(GRAPH(4,*)) :: G
  FINDMAX_Y_4 = MAXLOC(G%XVAL,1)
END FUNCTION FINDMAX_Y_4

INTEGER FUNCTION FINDMAX_Y_8(G)
  CLASS(GRAPH(8,*)) :: G
  FINDMAX_Y_8 = MAXLOC(G%XVAL,1)
END FUNCTION FINDMAX_Y_8
```

Final binding

Syntax of a *final_binding*

A derived type is finalizable if the derived type has any final subroutines or any nonpointer, nonallocatable component with a type that is finalizable. A nonpointer data entity is finalizable if the type of the entity is finalizable. The form of final_binding is:

```
►►──FINAL─────────────final_subroutine_name_list──────────────────►◄
          └─::─┘
```
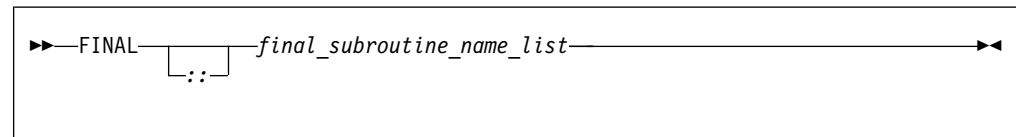
FINAL
Specifies a list of final subroutines. A final subroutine can be executed when a data entity of that type is finalized.

final_subroutine_name_list
A *final_subroutine_name* must be a module procedure with exactly one dummy argument. That argument must be nonoptional and must be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters are assumed; separate final subroutines must be defined for different kind parameters. The dummy argument cannot be **INTENT(OUT)**. A *final_subroutine_name* must not be one previously specified as a final subroutine for that type. A final subroutine must not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.

Final subroutines are not inherited through type extension and cannot be overridden.

The following is an example of extended types with final subroutines:

Example of extended types with final subroutines

```fortran
     MODULE m
     TYPE :: t1
        REAL a,b
     END TYPE
     TYPE, EXTENDS(t1) :: t2
        REAL, POINTER :: c(:), d(:)
      CONTAINS
        FINAL :: t2f
     END TYPE
     TYPE, EXTENDS(t2) :: t3 (k)
        INTEGER, KIND :: k
        REAL(k), POINTER :: e
      CONTAINS
        FINAL :: t3f4, t3f8
     END TYPE

CONTAINS
  SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
    TYPE(t2) :: x

    print *, 'entering t2f'
    IF (ASSOCIATED(x%c)) THEN
       print *, ' c allocated, cleaning up'
       DEALLOCATE(x%c)
    END IF
    IF (ASSOCIATED(x%d)) THEN
       print *, ' d allocated, cleaning up'
       DEALLOCATE(x%d)
    END IF
  END SUBROUTINE
  SUBROUTINE t3f4(y) ! Finalizer for TYPE(t3)'s extra components, where kind is 4
    TYPE(t3(4)) :: y
    print *, 'entering t3f4'
    IF (ASSOCIATED(y%e)) THEN
       print *, ' e (k=4) allocated, cleanup up'
       DEALLOCATE(y%e)
    END IF
  END SUBROUTINE
  SUBROUTINE t3f8(y) ! Second finalizer for TYPE(t3), with kind = 8
    TYPE(t3(8)) :: y
    print *, 'entering t3f8'
    IF (ASSOCIATED(y%e)) THEN
       print *, ' e (k=8) allocated, cleanup up'
       DEALLOCATE(y%e)
    END IF
  END SUBROUTINE
  ! If we had a type t3 with kind parameter k=16, we would probably
  ! want yet another finalizer with an argument of the appropriate
  ! type and parameters, but it does not have to be defined.
END MODULE

PROGRAM my_main
  CALL calc_span
END PROGRAM

SUBROUTINE calc_span
  USE m
  TYPE(t1) x1
  TYPE(t2) x2
  TYPE(t3(4)) x3
  TYPE(t3(8)) x3a

  ALLOCATE(x2%c(1:5), SOURCE=[1.0, 5.0, 10.0, 15.0, 20.0])
  ALLOCATE(x3%e, SOURCE=2.0)
  ALLOCATE(x3a%e, SOURCE=3.0_8)
```

```
      x2%c = x2%c + x3%e + x3a%e
      print *, 'calcs are=', x2%c

      ! Returning from this subroutine does
      ! nothing to x1. It is not finalizable.
      ! The Fortran compiler places calls to the finalizers at the
      ! end of a subroutine for the local variables of calc_span,
      ! as if the following calls were being made:
      ! CALL t2f(x2)
      ! CALL t3f4(x3)
      ! CALL t2f(x3%t2)
      ! CALL t3f8(x3a)
      ! CALL t2f(x3a%t2)
      ! Note that the specific order of invocation (x2 before x3 before x3a) may vary.
END SUBROUTINE
```

The output of the above program is:

calcs are= 6.000000000 10.00000000 15.00000000 20.00000000 25.00000000
 entering t2f
  c allocated, cleaning up
 entering t3f8
  e (k=8) allocated, cleanup up
 entering t2f
 entering t3f4
  e (k=4) allocated, cleanup up
 entering t2f

Related information
- "Finalization (Fortran 2003)" on page 68

Procedure overriding

If a nongeneric binding you specify in a type definition has the same binding name as a binding inherited from the parent type, then the binding you specify in the type definition overrides the binding inherited from the parent type.

The overriding binding and the overriden binding must satisfy the following conditions:
- Both bindings have a passed-object dummy argument or neither does.
- If the overridden binding is pure, the overriding binding must also be pure.
- Both bindings are elemental or neither is.
- Both bindings must have the same number of dummy arguments.
- Passed-object dummy arguments, if any, must correspond by name and position.
- Dummy arguments that correspond by position must have the same names and characteristics, except for the type of the passed-object dummy arguments.
- Both bindings must be subroutines or functions having the same result characteristics.
- If the overridden binding is **PUBLIC** then the overriding binding cannot be **PRIVATE**.

Example of procedure overriding
```
TYPE :: POINT
   REAL :: X, Y
   CONTAINS
   PROCEDURE, PASS :: LENGTH => POINT_LENGTH
END TYPE POINT
TYPE, EXTENDS (POINT) :: POINT_3D
```

```
        REAL :: Z
      CONTAINS
        PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
    END TYPE POINT_3D
    ...
```

The module-subprogram-part of the same module:
```
REAL FUNCTION POINT_LENGTH (A, B)
   CLASS (POINT), INTENT (IN) :: A, B
   POINT_LENGTH = SQRT ( (A%X - B%X)**2 + (A%Y - B%Y)**2 )
END FUNCTION POINT_LENGTH

REAL FUNCTION POINT_3D_LENGTH ( A, B )
   CLASS (POINT_3D), INTENT (IN) :: A
   CLASS (POINT), INTENT (IN) :: B
   SELECT TYPE(B)
      CLASS IS (POINT_3D)
      POINT_3D_LENGTH = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 + (A%Z-B%Z)**2 )
      RETURN
   END SELECT
   PRINT *, 'In POINT_3D_LENGTH, dynamic type of argument is incorrect.'
   STOP
END FUNCTION POINT_3D
```

If a generic binding specified in a type definition has the same *generic_spec* that does not satisfy the conditions as an inherited binding, it extends the generic interface and must satisfy the requirements specified in "Unambiguous generic procedure references" on page 168.

If a generic binding in a type definition has the same *dtio_generic_spec* as one inherited from the parent, it extends the generic interface for the *dtio_generic_spec* and must satisfy the requirements specified in "Unambiguous generic procedure references" on page 168.

A binding of a type and a binding of an extension of that type correspond if:
- The extension binding is the same as the type binding.
- The extension binding overrides a corresponding binding.
- The extension binding is an inherited corresponding binding.

Finalization (Fortran 2003)

The finalization process

Only finalizable entities are finalized. When an entity is finalized, the following steps are carried out in sequence:

1. If the dynamic type of the entity has a final subroutine and the dummy argument of the final subroutine has the same kind type parameters and rank as the entity being finalized, it is called with the entity as an actual argument. Otherwise, if an elemental final subroutine exists whose dummy argument has the same kind type parameters as the entity being finalized, it is called with the entity as an actual argument. No subroutine is called at this point otherwise.

2. Each finalizable component that appears in the type definition is finalized. If the entity being finalized is an array, each finalizable component of each element of that entity is finalized separately.

3. The parent component of an entity is finalized if the entity is of extended type and the parent type is finalizable.

If several entities are to be finalized as a result of one of the events that triggers finalization, these entities can be finalized in any order.

A final subroutine must not reference or define an object that has already been finalized. If the object is not finalized, it retains its definition status and is not undefined.

When finalization occurs

Finalization occurs for the target of a pointer when the pointer is deallocated. If an object is allocated through pointer allocation and later becomes unreachable because all pointers to that object have had their pointer association status changed, finalization on the object does not occur.

Finalization of an allocatable entity occurs when the entity is deallocated.

Finalization of a nonpointer, nonallocatable object that is not a dummy argument or function result occurs immediately before the object is undefined by the execution of a **RETURN** or **END** statement.

Finalization of the entity created by a structure constructor referenced by an executable construct occurs after execution of the innermost executable construct containing the reference.

Finalization of the entity created by an array constructor referenced by an executable construct occurs after execution of the innermost executable construct containing the reference.

Finalization of the result of a function referenced by an executable construct takes place after execution of the innermost executable construct containing the reference.

Finalization of the result of a function referenced by a specification expression in a scoping unit takes place before the first executable statement in the scoping unit executes.

Finalization of a nonpointer, nonallocatable object that is an actual argument associated with an **INTENT(OUT)** dummy argument of a procedure occurs when the procedure is invoked.

Finalization of a variable in an intrinsic assignment statement takes place after evaluation of the expression and before the definition of the variable.

`F2008` Finalization for an unsaved, nonpointer, nonallocatable, local variable of a **BLOCK** construct occurs immediately before execution exits the **BLOCK** construct. `F2008`

Non-finalized entities

If program execution is terminated, either by an error, such as an allocation failure, or by the execution of a **STOP**, `F2008` **ERROR STOP** `F2008` , or **END PROGRAM** statement, entities existing immediately before termination are not finalized.

A nonpointer, nonallocatable object that has the **SAVE** attribute is never finalized as a direct consequence of the execution of a **RETURN** or **END** statement.

Determining declared type for derived types

Two data objects have the same derived type if they are declared with reference to the same derived-type definition.

If the data objects are in different scoping units, they can still have the same derived type. Either the derived-type definition is accessible via host or use association, or the data objects reference their own derived-type definitions with the following conditions:

- They were both declared using standard derived type declarations, both have the same name, either both have the **SEQUENCE** property, or both have the **BIND** attribute, and both have components that do not have **PRIVATE** accessibility and their type parameter and components agree in order, name and attributes; or

- They were declared using record structure declarations that were not unnamed, the types have the same name, have no **%FILL** components and have components that agree in order and attributes, and any **%FILL** components appear in the same positions in both.

A derived-type definition that has the **BIND** attribute or the **SEQUENCE** property is not the same as a definition declared to be private or that has components that are private.

Examples

Example 1:

```
PROGRAM MYPROG

TYPE NAME                          ! Sequence derived type
   SEQUENCE
   CHARACTER(20) LASTNAME
   CHARACTER(10) FIRSTNAME
   CHARACTER(1)  INITIAL
END TYPE NAME
TYPE (NAME) PER1

CALL MYSUB(PER1)
PER1 = NAME('Smith','John','K')   ! Structure constructor
CALL MYPRINT(PER1)

CONTAINS
  SUBROUTINE MYSUB(STUDENT)        ! Internal subroutine MYSUB
     TYPE (NAME) STUDENT           ! NAME is accessible via host association
            ...
  END SUBROUTINE MYSUB
END

SUBROUTINE MYPRINT(NAMES)          ! External subroutine MYPRINT
  TYPE NAME                        ! Same type as data type in MYPROG
     SEQUENCE
     CHARACTER(20) LASTNAME
     CHARACTER(10) FIRSTNAME
     CHARACTER(1)  INITIAL
  END TYPE NAME
  TYPE (NAME) NAMES                ! NAMES and PER1 from MYPROG
  PRINT *, NAMES                   ! have the same data type
END SUBROUTINE
```

Example 2:

```
              MODULE MOD
                STRUCTURE /S/
                 INTEGER I
                 INTEGER, POINTER :: P
                END STRUCTURE
                RECORD /S/ R
              END MODULE
              PROGRAM P
                USE MOD, ONLY: R
                STRUCTURE /S/
                  INTEGER J
                  INTEGER, POINTER :: Q
                END STRUCTURE
                RECORD /S/ R2
                R = R2 ! OK - same type name, components have same attributes and
                       ! type (but different names)
              END PROGRAM P
```

Structure constructor

A structure constructor allows a scalar value of derived type to be constructed
from a list of values. A structure constructor must not appear before the definition
of the referenced derived type.
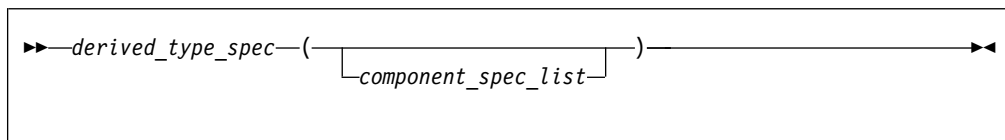
derived_type_spec
> is used to specify a particular derived type and type parameters.
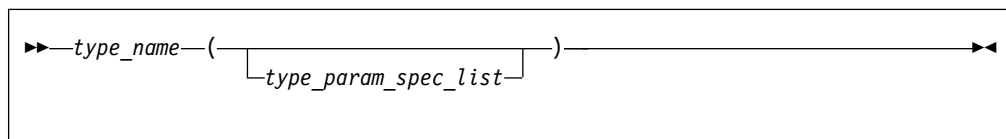
type_name
> is the name of the derived type, which must not be abstract.

type_param_spec is:
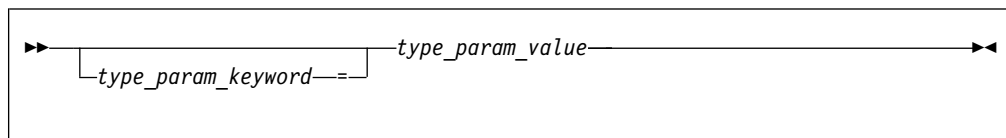
Note: The value of a type parameter for which no *type_param_value* has been
specified is its default value. For details, see "Derived type parameters (Fortran
2003)" on page 50.

type_param_keyword
> is the name of parameter of the type.

type_param_value
> must be a scalar integer expression in a structure constructor.

Note: `derived_type_spec` is also used in declaring data entities, procedure interfaces and so on. In these cases, a `type_param_value` that is a length parameter can be either a `*` or `:` in addition to scalar integer expression.

component_spec is:

```
>>─┬───────────────────────────┬─expr─────────────────────────────><
   └─component_keyword──=──┘
```
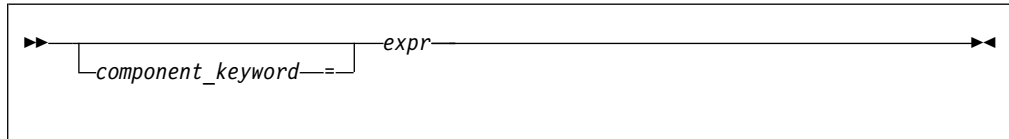
F2003 *component_keyword*
        is the name of a component of the type. F2003

expr   is an expression. Expressions are defined under Chapter 6, "Expressions and assignment," on page 101.

The *type_name* and all components of the type for which an *expr* appears must be accessible in the scoping unit containing the structure constructor.

In the absence of a component keyword, each *expr* is assigned to the corresponding component in component order. If a component keyword appears, the *expr* is assigned to the component named by the keyword. For a nonpointer component, the declared type and type parameters of the component and *expr* must conform in the same way as for a variable and expression in intrinsic assignment. If necessary, each value of intrinsic type is converted according to the rules of intrinsic assignment to a value that agrees in type and type parameters with the corresponding component of derived type. For a nonpointer nonallocatable component, the shape of the expression must conform with the shape of the component.

If a *component_spec* is provided for a component, no *component_spec* can be provided for any component with which it is inheritance-associated. At most one *component_spec* can be provided for a component.

If a component with default initialization has no corresponding *expr*, then the default initialization is applied to that component.

The *component_keyword* = specifier may be omitted from a *component_spec* only if the *component_keyword* = specifier has been omitted from each preceding *component_spec* in the constructor.

The *type_param_keyword* = specifier may be omitted from a *type_param_spec* only if the *type_param_keyword* = specifier has been omitted from each preceding *type_param_spec* in the constructor.

A component that is a pointer can be declared with the same type that it is a component of. If a structure constructor is created for a derived type containing a pointer, the expression corresponding to the pointer component must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement.

If a component of a derived type is allocatable, the corresponding constructor expression will either be a reference to the intrinsic function **NULL()** with no arguments, an allocatable entity, or will evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function **NULL()**, the corresponding

component of the constructor has a status of unallocated. If the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that of allocatable entity and, if it is allocated, it has the same dynamic type, bounds (if any) and value. If a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. Otherwise, the corresponding component of the constructor has an allocation status of allocated, and has the same bounds (if any) and value as the expression.

IBM If a derived type is declared using the **record structure** declaration and has any **%FILL** component, the structure constructor for that type cannot be used.

If a derived type is accessible in a scoping unit and there is a local entity of category 1 that is not a derived type with the same name accessible in the scoping unit, the structure constructor for that type cannot be used in that scope. **IBM**

If *derived_type_spec* is a type name that is the same as a generic name, *component_spec_list* must not be a valid *actual_arg_spec_list* for a function reference that is resolvable as a generic reference.

In a structure constructor of a derived type containing an allocatable component, the expression corresponding to the allocatable component must be one of the following:

- A reference to the intrinsic function NULL with no argument. The allocatable component receives the allocation status of unallocated.
- An allocatable variable with the same rank. The allocatable component has the allocation status of the variable and, if it is allocated, the same dynamic type, bounds, and value. The deferred length parameter of the component has the same value as the corresponding parameter of the variable.
- Any other expression that evaluates to an entity of the same rank. The allocatable component has the allocation status of allocated and has the same bounds and value as the expression.

Examples

Example 1:
```
MODULE PEOPLE
  TYPE NAME
    SEQUENCE                    ! Sequence derived type
    CHARACTER(20) LASTNAME
    CHARACTER(10) FIRSTNAME
    CHARACTER(1)  INITIAL
  END TYPE NAME

  TYPE PERSON                   ! Components accessible via use
                                ! association
    INTEGER AGE
    INTEGER BIRTHDATE(3)        ! Array component
    TYPE (NAME) FULLNAME        ! Component of derived type
  END TYPE PERSON
END MODULE PEOPLE

PROGRAM TEST1
  USE PEOPLE
  TYPE (PERSON) SMITH, JONES
  SMITH = PERSON(30, (/6,30,63/), NAME('Smith','John','K'))
                                ! Nested structure constructors
  JONES%AGE = SMITH%AGE         ! Component designator
  CALL TEST2
  CONTAINS
```

```
      SUBROUTINE TEST2
        TYPE T
          INTEGER EMP_NO
          CHARACTER, POINTER :: EMP_NAME(:)  ! Pointer component
        END TYPE T
        TYPE (T) EMP_REC
        CHARACTER, TARGET :: NAME(10)
        EMP_REC = T(24744,NAME)                 ! Pointer assignment occurs
      END SUBROUTINE                            ! for EMP_REC%EMP_NAME
END PROGRAM
```

Example 2:

```
PROGRAM LOCAL_VAR
  TYPE DT
    INTEGER A
    INTEGER :: B = 80
  END TYPE

  TYPE(DT) DT_VAR                   ! DT_VAR%B IS INITIALIZED
END PROGRAM LOCAL_VAR
```

Example 3:

```
MODULE MYMOD
  TYPE DT
    INTEGER :: A = 40
    INTEGER, POINTER :: B => NULL()
  END TYPE
END MODULE

PROGRAM DT_INIT
  USE MYMOD
  TYPE(DT), SAVE :: SAVED(8)        ! SAVED%A AND SAVED%B ARE INITIALIZED
  TYPE(DT) LOCAL(5)                 ! LOCAL%A LOCAL%B ARE INITIALIZED
END PROGRAM
```

`F2003`

Example 4:

```
PROGRAM NEW_LOCAL
  TYPE DT
    INTEGER :: A = 20
    INTEGER :: B = 80
  END TYPE

  TYPE(DT):: DT_VAR = DT()
  TYPE(DT):: DT_VAR2 = DT(B=40)
  TYPE(DT):: DT_VAR3 = DT(B=10, A=50)

  PRINT *, 'DT_VAR =', DT_VAR
  PRINT *, 'DT_VAR2=', DT_VAR2
  PRINT *, 'DT_VAR3=', DT_VAR3
END PROGRAM NEW_LOCAL

! The expeceted output is :
DT_VAR = 20 80
DT_VAR2= 20 40
DT_VAR3= 50 10
```

`F2003`

Related information:

"Local entity" on page 152

Chapter 5. Array concepts

An array is an ordered sequence of scalar data. All the elements of an array have the same type and type parameters.

XL Fortran provides a set of features, commonly referred to as array language, that allow you to manipulate arrays. This section provides background information on arrays and array language.

Many statements in Chapter 11, "Statements and attributes," on page 281, have special features and rules for arrays.

This section makes frequent use of the **DIMENSION** attribute. See "DIMENSION" on page 335.

A number of intrinsic functions are especially for arrays. These functions are mainly those classified as "Transformational intrinsic functions" on page 569.

Array basics

A *whole array* is denoted by the name of the array.

```
! In this declaration, the array is given a type and dimension
REAL, DIMENSION(3) :: A
! In these expressions, each element is evaluated in each expression
PRINT *, A, A+5, COS(A)
```

A whole array is either a named constant or a variable.

Dimension

In Fortran 2008, an array can have up to fifteen dimensions.

IBM In XL Fortran, an array can have up to twenty dimensions. IBM

Bounds of a dimension

Each dimension in an array has an upper and lower bound, which determine the range of values that can be used as subscripts for that dimension. The bound of a dimension can be positive, negative, or zero.

IBM In XL Fortran, the bound of a dimension can be positive, negative or zero within the range $-(2^{**}63)$ to $2^{**}63-1$. IBM

If any lower bound is greater than the corresponding upper bound, the array is a zero-sized array, which has no elements but still has the properties of an array. The return values for the intrinsic inquiry functions **LBOUND** and **UBOUND** for such a dimension are one and zero, respectively.

When the bounds are specified in array declarators:
• The lower bound is a specification expression. If it is omitted, the default value is 1.
• The upper bound is a specification expression or asterisk (*), and has no default value.

Related information
* "Specification expressions" on page 104

Extent of a dimension

The extent of a dimension is the number of elements in that dimension, computed as the value of the upper bound minus the value of the lower bound, plus one.

```
INTEGER, DIMENSION(5) :: X      ! Extent = 5
REAL :: Y(2:4,3:6)              ! Extent in 1st dimension = 3
                                ! Extent in 2nd dimension = 4
```

The minimum extent is zero, in a dimension where the lower bound is greater than the upper bound.

IBM The theoretical maximum number of elments in an array is 2**63-1 elements. Hardware addressing considerations make it impractical to declare any combination of data objects with a total size in bytes that exceeds this value. **IBM**

Different array declarators associated by common, equivalence, or argument association can have different ranks and extents.

Rank, shape, and size of an array
Rank

The rank of an array is the number of dimensions it has.

```
INTEGER, DIMENSION (10) :: A      ! Rank = 1
REAL, DIMENSION (-5:5,100) :: B  ! Rank = 2
```

A scalar is considered to have rank zero.

Shape

The shape of an array is derived from its rank and extents. It can be represented as a rank-one array where each element is the extent of the corresponding dimension:

```
INTEGER, DIMENSION (10,10) :: A          ! Shape = (/ 10, 10 /)
REAL, DIMENSION (-5:4,1:10,10:19) :: B   ! Shape = (/ 10, 10, 10 /)
```

Size

The size of an array is the total number of elements in it. The size equals to the product of the extents of all dimensions.

```
INTEGER A(5)               ! Size = 5
REAL B(-1:0,1:3,4)         ! Size = 2 * 3 * 4 = 24
```

Array declarators

An array declarator specifies the rank or rank and shape of an array.

You must declare every named array, and no scoping unit can have more than one array declarator for the same name. An array declarator can appear in any of the *Compatible Statements and Attributes for Array Declarators* table.

Table 14. Compatible statements and attributes for array declarators

ALLOCATABLE **1**	AUTOMATIC **2**	COMMON
DIMENSION	CONTIGUOUS **3**	PARAMETER

Table 14. Compatible statements and attributes for array declarators (continued)

POINTER (integer) **2**	POINTER	PROTECTED **1**
STATIC **2**	TARGET	Type Declaration
VOLATILE		
Notes: **1** Fortran 2003 **2** IBM extension **3** Fortran 2008		

For example:

```
DIMENSION :: A(1:5)         ! Declarator is "(1:5)"
REAL, DIMENSION(1,1:5) :: B ! Declarator is "(1,1:5)"
INTEGER C(10)               ! Declarator is "(10)"
```

The form of an array declarator is:
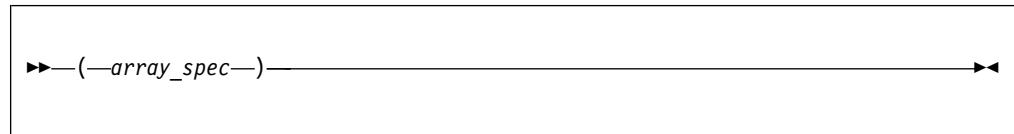
array_spec

> is an array specification. It is a list of dimension declarators, each of which establishes the lower and upper bounds of an array, or specifies that one or both will be set at run time. Each dimension requires one dimension declarator.

> An *array_spec* is one of:
> *explicit_shape_spec_list*
> *assumed_shape_spec_list*
> *deferred_shape_spec_list*
> *implied_shape_spec_list*
> *assumed_size_spec*
> TS 29113 *assumed_rank_spec* TS 29113

> Each *array_spec* declares a different kind of array. For details, see the topics in the Chapter 5, "Array concepts," on page 75 section.

Explicit-shape arrays

Explicit-shape arrays are arrays where the bounds are explicitly specified for each dimension.

Explicit_shape_spec_list
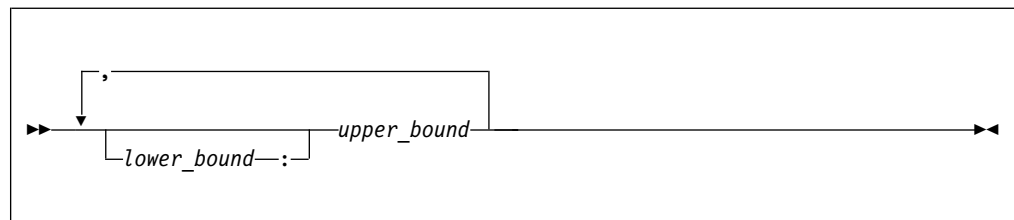
lower_bound, upper_bound
> are specification expressions

Arrays with bounds that are nonconstant expressions must be declared inside subprograms or ►F2008◄ **BLOCK** constructs ►F2008◄ . The nonconstant bounds are determined on entry to the subprogram or ►F2008◄ **BLOCK** construct ►F2008◄ . If a lower bound is omitted, its default value is one.

The rank is the number of specified upper bounds. The shape of an explicit-shape dummy argument can differ from that of the corresponding actual argument.

The size is determined by the specified bounds.

The size of an explicit-shape dummy argument does not need to be the same as the size of the actual argument, but the size of the dummy argument cannot be larger than the size of the actual argument.

►F2008◄ You can specify the **VALUE** attribute on explicit-shape arrays. ►F2008◄

Examples

```
INTEGER A,B,C(1:10,-5:5)  ! All bounds are constant
A=8; B=3
CALL SUB1(A,B,C)
END
SUBROUTINE SUB1(X,Y,Z)
  INTEGER X,Y,Z(X,Y)      ! Some bounds are not constant
END SUBROUTINE
```

Related information

"BLOCK construct (Fortran 2008)" on page 136

"VALUE (Fortran 2003)" on page 487

Automatic arrays

An automatic array is an explicit-shape array that is declared in a subprogram ►F2008◄ or a **BLOCK** construct ►F2008◄ . It is not a dummy argument or pointee array, and has at least one bound that is a nonconstant specification expression. Evaluation of the bounds occurs on entry into the subprogram ►F2008◄ or **BLOCK** construct ►F2008◄ . After the bounds are determined, they remain unchanged during execution of the subprogram ►F2008◄ or **BLOCK** construct ►F2008◄ .

Example of an automatic array

```
INTEGER X
COMMON X
X = 10
CALL SUB1(5)
END

SUBROUTINE SUB1(Y)
  INTEGER X
  COMMON X
  INTEGER Y
  REAL Z (X:20, 1:Y)        ! Automatic array.  Here the bounds are made
                            ! available through dummy arguments and common
                            ! blocks, although Z itself is not a dummy
END SUBROUTINE              ! argument.
```

Related information
- For general information about automatic data objects, see "Automatic objects" on page 20 and "Storage classes for variables (IBM extension)" on page 28.

Adjustable arrays

An adjustable array is an explicit-shape array dummy argument that has at least one non-constant bound.

Example of an adjustable array
```
SUBROUTINE SUB1(X, Y)
INTEGER X, Y(X*3)  ! Adjustable array.  Here the bounds depend on a
                   ! dummy argument, and the array name is also passed in.
END SUBROUTINE
```

Pointee arrays (IBM extension)

Pointee arrays are explicit-shape or assumed-size arrays that can only appear in integer **POINTER** statements.

The declarator for a pointee array can only contain variables if you declare the array inside a subprogram, and any such variables must be:
- dummy arguments
- members of a common block
- use associated
- host associated

Evaluation of the bounds occurs on entry into the subprogram, and remain constant during execution of that subprogram.

Compiling with the **-qddim option** option relaxes the restrictions on which variables can appear in an array declarator. Declarators in the main program can contain variable names, and any specified nonconstant bounds are re-evaluated each time you reference the array, so that you can change the properties of the pointee array by changing the values of the variables used in the bounds expressions.

Example using -qddim to relax array declarator restrictions
```
@PROCESS DDIM
INTEGER PTE, N, ARRAY(10)
POINTER (P, PTE(N))
N = 5
P = LOC(ARRAY(2))  !
PRINT *, PTE       ! Print elements 2 through 6 of ARRAY
N = 7              ! Increase the size
PRINT *, PTE       ! Print elements 2 through 8 of ARRAY
END
```

Assumed-shape arrays

Assumed-shape arrays are dummy argument arrays where the extent of each dimension is taken from the associated actual arguments.

Assumed_shape_spec_list

lower_bound
        is a specification expression

Each lower bound defaults to one, or may be explicitly specified. Each upper bound is set on entry to the subprogram to the specified lower bound (not the lower bound of the actual argument array) plus the extent of the dimension minus one.

The extent of any dimension is the extent of the corresponding dimension of the associated actual argument.

The rank is the number of colons in the *assumed_shape_spec_list*.

The shape is assumed from the associated actual argument array.

The size is determined on entry to the subprogram where it is declared, and equals the size of the associated actual argument array.

Note: Subprograms that have assumed-shape arrays as dummy arguments must have explicit interfaces.

F2008 You can specify the **VALUE** attribute on assumed-shape arrays. **F2008**

Examples

```
INTERFACE
  SUBROUTINE SUB1(B)
    INTEGER B(1:,:,10:)
  END SUBROUTINE
END INTERFACE
INTEGER A(10,11:20,30)
CALL SUB1 (A)
END
SUBROUTINE SUB1(B)
INTEGER B(1:,:,10:)
! Inside the subroutine, B is associated with A.
! It has the same extents as A but different bounds (1:10,1:10,10:39).
END SUBROUTINE
```

Related information

"BLOCK construct (Fortran 2008)" on page 136

"VALUE (Fortran 2003)" on page 487

Deferred-shape arrays

Deferred-shape arrays are allocatable arrays or array pointers, where the bounds can be defined or redefined during execution of the program.

Deferred_shape_spec_list

```
          ┌ , ┐
          │   │
▶▶──┬──────▼──┴──────────────────────────────────────────────────▶◀
    └─:──┘
```

The extent of each dimension (and the related properties of bounds, shape, and size) is undefined until the array is allocated or the pointer is associated with an array that is defined. Before then, no part of the array may be defined, or referenced except as an argument in certain situations.

An array pointer can have its bounds specified by one of the following:
- pointer assignment
- **ALLOCATE** statement
- if it is a dummy argument, argument association with a non-pointer actual argument or an associated pointer argument

An allocatable array can have its bounds specified by one of the following:
- **ALLOCATE** statement
- if it is a dummy argument, argument association with an allocated argument

The rank is the number of colons in the *deferred_shape_spec_list*.

Although a *deferred_shape_spec_list* can appear identical to an *assumed_shape_spec_list*, deferred-shape arrays and assumed-shape arrays are not the same. A deferred-shape array must have the **ALLOCATABLE** or **POINTER** attribute, while an assumed-shape array must be a dummy argument that does not have the **ALLOCATABLE** or **POINTER** attribute. The bounds of a deferred-shape array, and the actual storage associated with it, can be changed at any time by reallocating the array or by associating the pointer with a different array, while these properties remain the same for an assumed-shape array during the execution of the containing subprogram.

Related information:
- "Allocation status" on page 28
- "Data pointer assignment" on page 130
- "Pointer association" on page 159
- "ALLOCATABLE (Fortran 2003)" on page 285
- "ALLOCATED(X)" on page 580
- "ASSOCIATED(POINTER, TARGET)" on page 585

Allocatable arrays

A deferred-shape array that has the **ALLOCATABLE** attribute is referred to as an allocatable array. The bounds and shape of the array are determined when you perform one of the following tasks:
- Allocate storage using an **ALLOCATE** statement
- Perform argument association

Example

The following example declares an allocatable array and determines its bounds.

```
INTEGER, ALLOCATABLE, DIMENSION(:, :, :) :: arr
ALLOCATE(arr(10, -4:5, 20))   ! Bounds of arr are now defined (1:10, -4:5, 1:20)
DEALLOCATE(a)
ALLOCATE(arr(5, 5, 5))        ! Change the bounds of arr
```

If you compile your program with **-qinitalloc**, all elements of the allocatable array arr are initialized to zero.

Migration Tip:

If you do not know the size of an array at compile time, you can avoid unnecessary memory usage by making the array allocatable instead of declaring it with a maximum size.

FORTRAN 77 source

```
      INTEGER A(1000),B(1000),C(1000)
C 1000 is the maximum size
      WRITE (6,*) "Enter the size of the arrays:"
      READ (5,*) N
 .
 .
 .
      DO I=1,N
        A(I)=B(I)+C(I)
      END DO
      END
```

Source for Fortran 90 or above:

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A,B,C
WRITE (6,*) "Enter the size of the arrays:"
READ (5,*) N
ALLOCATE (A(N),B(N),C(N))
 .
 .
 .
A=B+C
END
```

Related information
* "ALLOCATABLE (Fortran 2003)" on page 285
* "ALLOCATE" on page 287
* "DEALLOCATE" on page 331
* The **-qinitalloc** option

Array pointers

An array with the **POINTER** attribute is referred to as an array pointer. Its bounds and shape are determined when it is associated with a target through the following ways:
* Pointer assignment
* Argument association
* Execution of an **ALLOCATE** statement

Example

The following example declares an array pointer and determines its bounds and storage association.

```
REAL, POINTER, DIMENSION(:, :) :: b
REAL, TARGET, DIMENSION(5, 10) :: c, d(10, 10)
b => c   ! Bounds of b are now defined (1:5, 1:10)
b => d   ! b now has different bounds and is associated with different storage
```

If you use the following **ALLOCATE** statement and compile your program with the **-qinitalloc** option, all elements of the array pointer b are initialized to zero.

```
ALLOCATE(b(5, 5)) ! Change bounds and storage association again
```

Related information
- "Pointer association" on page 159
- "ALLOCATE" on page 287
- The **-qinitalloc** option

Implied-shape arrays (Fortran 2008)

An implied-shape array is a named constant that inherits its shape from the constant expression in its declaration.

Implied_shape_spec_list
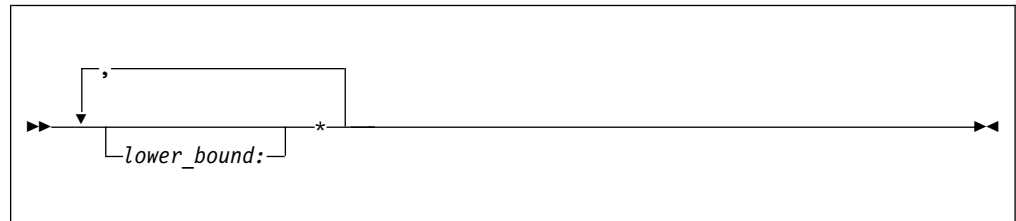
lower_bound
>A specification expression

The declaration of an implied-shape array contains an implied-shape specification and a constant expression. The constant expression must be an array.

The rank is the number of implied-shape specifications in *implied_shape_spec_list*.

The extent of any dimension is the same as the extent of the corresponding dimension of the constant expression.

Each lower bound is the corresponding lower bound in *implied_shape_spec_list*. For dimensions whose lower bounds are not specified, the lower bounds default to one. Each upper bound is the sum of the lower bound and extent minus one.

Examples

```
! Array imp1 is a rank-one array. Its upper bound is 5.
INTEGER, PARAMETER :: imp1(4:*) = [1, 2]

! Array imp2 is a rank-one array. Its upper bound is 4.
INTEGER, PARAMETER :: onetofour(4) = [1, 2, 3, 4]
INTEGER, PARAMETER, DIMENSION(*) :: imp2 = onetofour
```

```
! Array imp3 is a rank-two array. Its shape is (/2, 2/) and the upper bounds for
the two dimensions are 2 and 11.
REAL, PARAMETER :: imp3(*, 10:*) = RESHAPE([1, 2, 3, 4], [2, 2])
```

Related information
- Constant expressions in the *XL Fortran Language Reference*

Assumed-size arrays

Assumed-size arrays are dummy argument arrays where the size is inherited from the associated actual argument, but the rank and extents might differ.

Assumed_size_spec
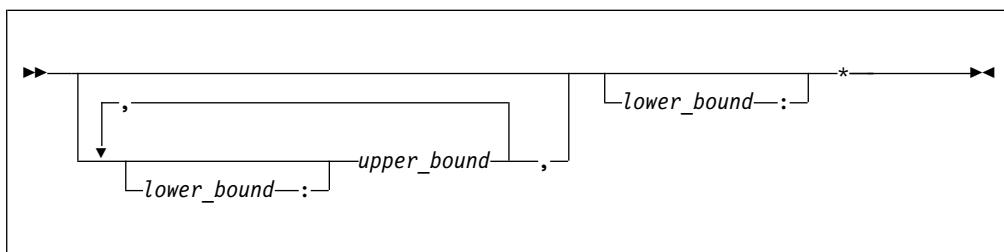
lower_bound, upper_bound
> are specification expressions

If any bound is not constant, the array must be declared inside a subprogram and the nonconstant bounds are determined on entry to the subprogram. If a lower bound is omitted, its default value is 1.

The last dimension has no upper bound and is designated instead by an asterisk. You must ensure that references to elements do not go past the end of the actual array.

The rank equals one plus the number of *upper_bound* specifications in its declaration, which may be different from the rank of the actual array it is associated with.

The size is assumed from the actual argument that is associated with the assumed-size array:

- TS 29113 If the actual argument is a scalar and the assumed-size dummy argument is of assumed-type, the assumed-size array has exactly one element. TS 29113
- If the actual argument is a noncharacter array, the size of the assumed-size array is that of the actual array.
- If the actual argument is an array element from a noncharacter array, and if the size remaining in the array beginning at this element is **S**, then the size of the dummy argument array is **S**. Array elements are processed in array element order.
- If the actual argument is a character array, array element, or array element substring, and assuming that:
  - **A** is the starting offset, in characters, into the character array
  - **T** is the total length, in characters, of the original array
  - **S** is the length, in characters, of an element in the dummy argument array

  then the size of the dummy argument array is:

MAX(INT (T - A + 1) / S, 0)

For example:

```
CHARACTER(10) A(10)
CHARACTER(1) B(30)
CALL SUB1(A)          ! Size of dummy argument array is 10
CALL SUB1(A(4))       ! Size of dummy argument array is 7
CALL SUB1(A(6)(5:10)) ! Size of dummy argument array is 4 because there
                      ! are just under 4 elements remaining in A
CALL SUB1(B(12))      ! Size of dummy argument array is 1, because the
                      ! remainder of B can hold just one CHARACTER(10)
END                   ! element.
SUBROUTINE SUB1(ARRAY)
  CHARACTER(10) ARRAY(*)
  ...
END SUBROUTINE
```

Examples

Example 1

```
INTEGER X(3,2)
DO I = 1,3
   DO J = 1,2
      X(I,J) = I * J      ! The elements of X are 1, 2, 3, 2, 4, 6
   END DO
END DO
PRINT *,SHAPE(X)        ! The shape is (/ 3, 2 /)
PRINT *,X(1,:)          ! The first row is (/ 1, 2 /)
CALL SUB1(X)
CALL SUB2(X)
END
SUBROUTINE SUB1(Y)
  INTEGER Y(2,*)        ! The dimensions of y are the reverse of x above
  PRINT *, SIZE(Y,1)    ! We can examine the size of the first dimension
                        ! but not the last one.
  PRINT *, Y(:,1)       ! We can print out vectors from the first
  PRINT *, Y(:,2)       ! dimension, but not the last one.
END SUBROUTINE
SUBROUTINE SUB2(Y)
  INTEGER Y(*)          ! Y has a different rank than X above.
  PRINT *, Y(6)         ! We have to know (or compute) the position of
                        ! the last element.  Nothing prevents us from
                        ! subscripting beyond the end.
END SUBROUTINE
```

TS 29113

Example 2

```
REAL r1
CALL sub(r1)
!You can pass a scalar as the actual argument that corresponds
!to an assumed-size dummy argument of assumed-type.
CONTAINS
  SUBROUTINE sub(arg)
    TYPE(*) :: arg(3, 2:*)
  END
END
```

TS 29113

Notes:

1. An assumed-size array cannot be used as a whole array in an executable construct unless it is an actual argument in a subprogram reference that does not require the shape:

```
! A is an assumed-size array.
PRINT *, UBOUND(A,1) ! OK - only examines upper bound of first dimension.
PRINT *, LBOUND(A)   ! OK - only examines lower bound of each dimension.
! However, 'B=UBOUND(A)' or 'A=5' would reference the upper bound of
! the last dimension and are not allowed.  SIZE(A) and SHAPE(A) are
! also not allowed.
```

2. If a section of an assumed-size array has a subscript triplet as its last section subscript, the upper bound must be specified. (Array sections and subscript triplets are explained in a subsequent section.)

```
! A is a 2-dimensional assumed-size array
PRINT *, A(:, 6)       ! Triplet with no upper bound is not last dimension.
PRINT *, A(1, 1:10)    ! Triplet in last dimension has upper bound of 10.
PRINT *, A(5, 5:9:2)   ! Triplet in last dimension has upper bound of 9.
```

Assumed-rank objects (TS 29113)

Assumed-rank objects are dummy argument objects whose ranks are assumed from the associated actual arguments at run time. The actual argument that corresponds to an assumed-rank object can be an array or a scalar.

Assumed_rank_spec

```
►►── .. ──────────────────────────────────────────►◄
```

If an assumed-rank dummy argument is associated with a scalar actual argument, the rank of the assumed-rank dummy argument is zero, and the shape of it is a zero-sized array.

Restrictions:

- An assumed-type actual argument that corresponds to an assumed-rank dummy argument must be assumed-shape or assumed-rank.
- An assumed-rank object cannot have the **VALUE** attribute.
- An assumed-rank object cannot be used in a designator or expression except in the following ways:
  - As an actual argument that corresponds to a dummy argument that is assumed-rank
  - As the argument of the **C_LOC** function in the **ISO_C_BINDING** intrinsic module
  - As the first argument in a reference to an intrinsic inquiry function

Examples

```
REAL :: a0
REAL :: a1(10)
REAL :: a2(10, 20)
REAL, POINTER :: a3(:,:,:)

CALL sub1(a0)
CALL sub1(a1)
CALL sub1(a2)
```

```
CALL sub1(a3)

CONTAINS
  SUBROUTINE sub1(a)
    REAL :: a(..)
    PRINT *, RANK(a)
  END

END
```

The output is as follows:
```
0
1
2
3
```

Related information
- "Contiguity (Fortran 2008)" on page 98
- "LBOUND(ARRAY, DIM, KIND)" on page 642
- "RANK(A) (TS 29113)" on page 688
- "UBOUND(ARRAY, DIM, KIND)" on page 720
- "VALUE (Fortran 2003)" on page 487
- "Interoperability of assumed-rank arguments (TS 29113)" on page 897

Array elements

Array elements are the scalar data that make up an array. Each element inherits the type, type parameters, and **INTENT**, **PARAMETER**, **TARGET**, F2003 **PROTECTED**, **ASYNCHRONOUS**, and **VOLATILE** F2003 attributes from its parent array. The **POINTER** and **ALLOCATABLE** attributes are not inherited.

Syntax

You identify an array element by an *array element designator*, whose form is:

```
►►──┬─array_name────────┬──(─subscript_list─)──────────────────►◄
    └─array_struct_comp─┘
```
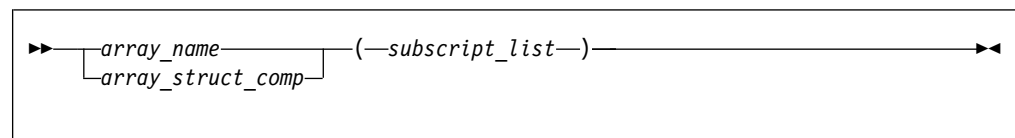
array_name
> is the name of an array

array_struct_comp
> is a structure component whose rightmost *comp_name* is an array

subscript
> is a scalar integer expression
>
> IBM A subscript can be a scalar real expression in XL Fortran. IBM

Rules
- The number of subscripts must equal the number of dimensions in the array.
- If *array_struct_comp* is present, each part of the structure component except the rightmost must have rank zero (that is, must not be an array name or an array section).
- The value of each subscript expression must not be less than the lower bound or greater than the upper bound for the corresponding dimension.

- The *subscript* value depends on the value of each subscript expression and on the dimensions of the array. It determines which element of the array is identified by the array element designator.

Array element order

The elements of an array are arranged in storage in a sequence known as the *array element order*, in which the subscripts change most rapidly in the first dimension, and subsequently in the remaining dimensions.

For example, an array declared as A(2, 3, 2) has the following elements:

```
Position of Array Element          Array Element Order
-------------------------          -------------------
         A(1,1,1)                          1
         A(2,1,1)                          2
         A(1,2,1)                          3
         A(2,2,1)                          4
         A(1,3,1)                          5
         A(2,3,1)                          6
         A(1,1,2)                          7
         A(2,1,2)                          8
         A(1,2,2)                          9
         A(2,2,2)                         10
         A(1,3,2)                         11
         A(2,3,2)                         12
```

Related information
"Derived type components" on page 51
"Array sections and structure components" on page 92

Array sections

An array section is a selected portion of an array. It is an array subobject that designates a set of elements from an array, or a specified substring or derived-type component from each of those elements. An array section is also an array.

Note: This introductory section describes the simple case, where structure components are not involved. "Array sections and structure components" on page 92 explains the additional rules for specifying array sections that are also structure components.

```
►►──array_name──(──section_subscript_list──)─┬──────────────────┬──►◄
                                             └─substring_range─┘
```
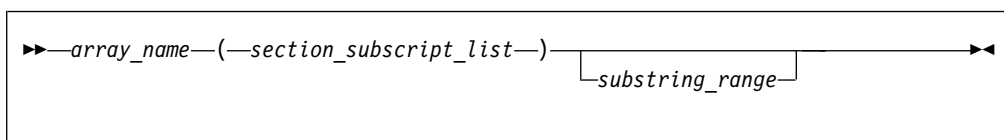
section_subscript

        designates some set of elements along a particular dimension. It can be composed of a combination of the following:

```
►►──┬──subscript──────────┬──►◄
    ├──subscript_triplet──┤
    └──vector_subscript───┘
```
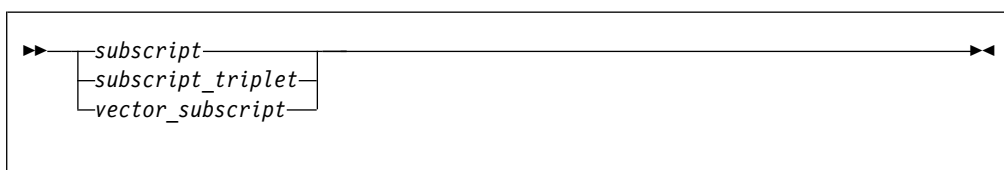
subscript

> is a scalar integer expression. For details, see "Array elements" on page 87.
>
> IBM A subscript can be a scalar real expression in XL Fortran. IBM

subscript_triplet, *vector_subscript*

> designate a sequence of subscripts in a given dimension. For details, see "Subscript triplets" on page 90 and "Vector subscripts" on page 91.

Notes:
- At least one of the dimensions must be a subscript triplet or vector subscript, so that an array section is distinct from an array element. See Example 1.
- An array section can contain a set of array elements that is contiguous or not contiguous within the array. For more information, see "Contiguity (Fortran 2008)" on page 98.

substring_range

```
►►──(──┬─────────────┬──:──┬─────────────┬──)────────────────►◄
        └─ int_expr1 ─┘      └─ int_expr2 ─┘
```
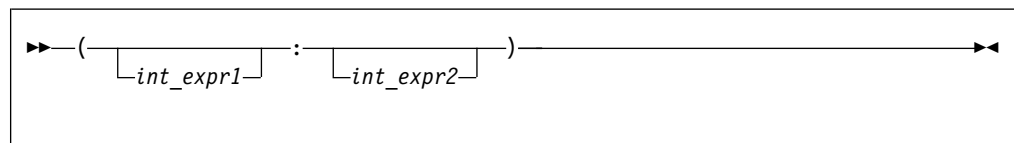
int_expr1, *int_expr2*

> are scalar integer expressions called substring expressions, defined in "Character substrings" on page 46. They specify the leftmost and rightmost character positions, respectively, of a substring of each element in the array section. If an optional *substring_range* is present, the section must be from an array of character objects. For details, see "Substring ranges" on page 92.

An array section is formed from the array elements specified by the sequences of values from the individual subscripts, subscript triplets, and vector subscripts, arranged in column-major order. See Example 2.

Examples

Example 1
```
INTEGER, DIMENSION(5,5,5) :: A
A(1,2,3) = 100
A(1,3,3) = 101
PRINT *, A(1,2,3)     ! A single array element, 100.
PRINT *, A(1,2:2,3)   ! A one-element array section, (/ 100 /)
PRINT *, A(1,2:3,3)   ! A two-element array section,
                      ! (/ 100, 101 /)
```

Example 2

If `SECTION = A(1:3, (/5, 6, 5/), 4)`
- The sequence of numbers for the first dimension is 1, 2, 3.
- The sequence of numbers for the second dimension is 5, 6, 5.
- The subscript for the third dimension is the constant 4.

The section is made up of the following elements of A, in this order:

```
A(1,5,4)    |                       |    SECTION(1,1)
A(2,5,4)    |----- First column ----|    SECTION(2,1)
A(3,5,4)    |                       |    SECTION(3,1)
A(1,6,4)    |                       |    SECTION(1,2)
A(2,6,4)    |----- Second column ---|    SECTION(2,2)
A(3,6,4)    |                       |    SECTION(3,2)
A(1,5,4)    |                       |    SECTION(1,3)
A(2,5,4)    |----- Third column ----|    SECTION(2,3)
A(3,5,4)    |                       |    SECTION(3,3)
```

Example 3

Some other examples of array sections include:

```
INTEGER, DIMENSION(20,20) :: A
! These references to array sections require loops or multiple
! statements in FORTRAN 77.
PRINT *, A(1:5,1)                   ! Contiguous sequence of elements
PRINT *, A(1:20:2,10)               ! Noncontiguous sequence of elements
PRINT *, A(:,5)                     ! An entire column
PRINT *, A( (/1,10,5/), (/7,3,1/) ) ! A 3x3 assortment of elements
```

Related information
• "Derived type components" on page 51

Subscript triplets

A subscript triplet consists of two subscripts and a stride, and defines a sequence of numbers corresponding to array element positions along a single dimension.

```
►►─┬─────────────┬─:─┬─────────────┬─┬──────────────┬─────►◄
   └─subscript1──┘   └─subscript2──┘ └─:──stride─────┘
```
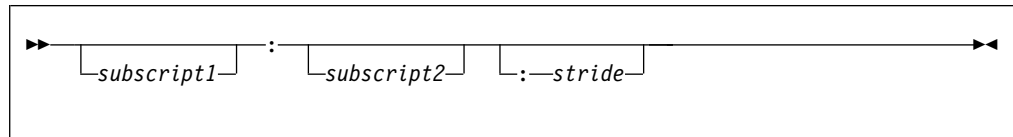
subscript1
> is the subscript that designates the first value in the sequence of indices for a dimension.
>
> If it is omitted, the lower array bound of that dimension is used.

subscript2
> is the subscript that designates the last value in the sequence of indices for a dimension.
>
> If it is omitted, the upper array bound of that dimension is used. It is mandatory for the last dimension when specifying sections of an assumed-size array.

stride is a scalar integer expression that specifies how many subscript positions to count to reach the next selected element.

> **IBM** A stride can be a scalar real expression in XL Fortran. **IBM**
>
> If the stride is omitted, it has a value of 1. The stride must have a nonzero value:
>
> • A positive stride specifies a sequence of integers that begins with the first subscript and proceeds in increments of the stride to the largest integer that is not greater than the second subscript. If the first subscript is greater than the second, the sequence is empty.

- When the stride is negative, the sequence begins at the first subscript and continues in increments specified by the stride to the smallest integer equal to or greater than the second subscript. If the second subscript is greater than the first, the sequence is empty.

Calculations of values in the sequence use the same steps as shown in "Executing a DO statement" on page 140.

A subscript in a subscript triplet does not have to be within the declared bounds for that dimension if all the values used in selecting the array elements for the array section are within the declared bounds:

```
INTEGER A(9)
PRINT *, A(1:9:2)  ! Count from 1 to 9 by 2s: 1, 3, 5, 7, 9.
PRINT *, A(1:10:2) ! Count from 1 to 10 by 2s: 1, 3, 5, 7, 9.
                   ! No element past A(9) is specified.
```

Examples

```
REAL, DIMENSION(10) :: A
INTEGER, DIMENSION(10,10) :: B
CHARACTER(10) STRING(1:100)

PRINT *, A(:)             ! Print all elements of array.
PRINT *, A(:5)            ! Print elements 1 through 5.
PRINT *, A(3:)            ! Print elements 3 through 10.

PRINT *, STRING(50:100)   ! Print all characters in
                          ! elements 50 through 100.

! The following statement is equivalent to A(2:10:2) = A(1:9:2)
A(2::2) = A(:9:2)         ! LHS = A(2), A(4), A(6), A(8), A(10)
                          ! RHS = A(1), A(3), A(5), A(7), A(9)
                          ! The statement assigns the odd-numbered
                          ! elements to the even-numbered elements.

! The following statement is equivalent to PRINT *, B(1:4:3,1:7:6)
PRINT *, B(:4:3,:7:6)     ! Print B(1,1), B(4,1), B(1,7), B(4,7)

PRINT *, A(10:1:-1)       ! Print elements in reverse order.

PRINT *, A(10:1:1)        ! These two are
PRINT *, A(1:10:-1)       ! both zero-sized.
END
```

Vector subscripts

A vector subscript is an integer array expression of rank one, designating a sequence of subscripts that correspond to the values of the elements of the expression.

IBM A vector subscript can be a real array expression of rank one in XL Fortran.
IBM

The sequence does not have to be in order, and may contain duplicate values:

```
INTEGER A(10), B(3), C(3)
PRINT *, A( (/ 10,9,8 /) ) ! Last 3 elements in reverse order
B = A( (/ 1,2,2 /) )       ! B(1) = A(1), B(2) = A(2), B(3) = A(2) also
END
```

If a vector subscript has more than one element with the same value, an array section with that vector subscript cannot appear in any of the situations mentioned in "Syntactic contexts of definition or undefinition of variables" on page 27.

Notes:

1. An array section used as an internal file must not have a vector subscript.
2. If you pass an array section with a vector subscript as an actual argument, the associated dummy argument must not be defined or redefined.
3. An array section with a vector subscript must not be the target in a pointer assignment statement.
4. `F2008` In XL Fortran, a nonzero-sized array section containing a vector subscript is considered noncontiguous. For details, see Contiguity. `F2008`

```
! We can use the whole array VECTOR as a vector subscript for A and B
INTEGER, DIMENSION(3) :: VECTOR= (/ 1,3,2 /), A, B
INTEGER, DIMENSION(4) :: C = (/ 1,2,4,8 /)
A(VECTOR) = B            ! A(1) = B(1), A(3) = B(2), A(2) = B(3)
A = B( (/ 3,2,1 /) )     ! A(1) = B(3), A(2) = B(2), A(3) = B(1)
PRINT *, C(VECTOR(1:2))  ! Prints C(1), C(3)
END
```

Substring ranges

For an array section with a substring range, each element in the result is the designated character substring of the corresponding element of the array section. The rightmost array name or component name must be of type character.

```
PROGRAM SUBSTRING
TYPE DERIVED
  CHARACTER(10) STRING(5)       ! Each structure has 5 strings of 10 chars.
END TYPE DERIVED
TYPE (DERIVED) VAR, ARRAY(3,3)  ! A variable and an array of derived type.

VAR%STRING(:)(1:3) = 'abc'      ! Assign to chars 1-3 of elements 1-5.
VAR%STRING(3:)(4:6) = '123'     ! Assign to chars 4-6 of elements 3-5.

ARRAY(1:3,2)%STRING(3)(5:10) = 'hello'
                                ! Assign to chars 5-10 of the third element in
                                ! ARRAY(1,2)%STRING, ARRAY(2,2)%STRING, and
END                             ! ARRAY(3,2)%STRING
```

Array sections and structure components

Understanding how array sections and structure components interact requires a familiarity with the syntax for "Derived type components" on page 51.

What we defined at the beginning of this section as an array section is really only a subset of the possible array sections. An array name or array name with a *section_subscript_list* can be a subobject of a structure component:

struct_sect_subobj:

```
>>--object_name----------------------------------------------->
              └-(--section_subscript_list--)-┘

   ┌<--------------------------------------------------┐
>--┴-┬-%-┬---comp_name------------------------------------------->
      └-.-┘           └-(--section_subscript_list--)-┘

>--┬-----------------------┬--><
   └-substring_range-┘
```
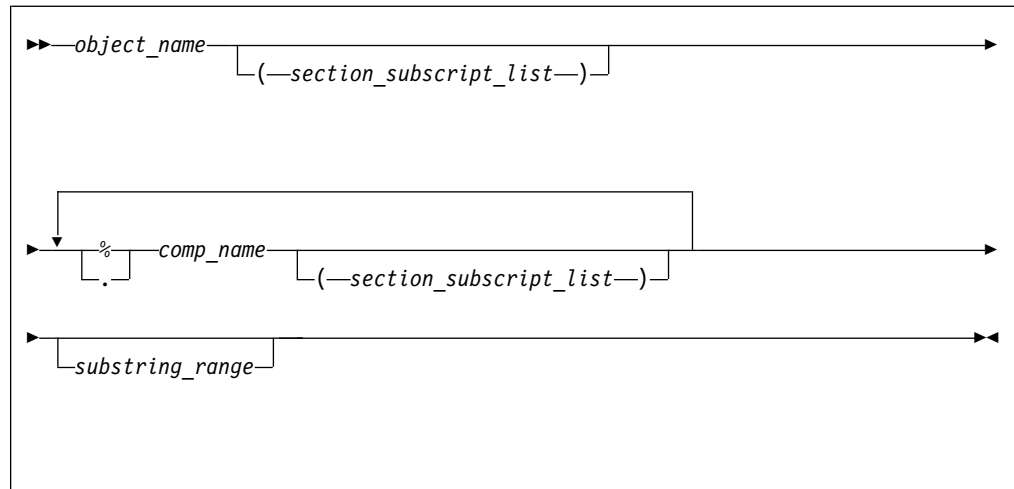
object_name
        is the name of an object of derived type

section_subscript_list, substring_range
        are the same as defined under "Array sections" on page 88

comp_name
        is the name of a derived-type component

% or . Separator character.

      Note: The **.** (period) separator is an IBM extension.

Note:

1. The type of the last component determines the type of the array.
2. Only one part of the structure component may have nonzero rank. Either the rightmost *comp_name* must have a *section_subscript_list* with nonzero rank, or another part must have nonzero rank.
3. Any parts to the right of the part with nonzero rank must not have the **ALLOCATABLE** or **POINTER** attributes.

```
TYPE BUILDING_T
  LOGICAL RESIDENTIAL
END TYPE BUILDING_T

TYPE STREET_T
  TYPE (BUILDING_T) ADDRESS(500)
END TYPE STREET_T

TYPE CITY_T
  TYPE (STREET_T) STREET(100,100)
END TYPE CITY_T

TYPE (CITY_T) PARIS
TYPE (STREET_T) S
TYPE (BUILDING_T) RESTAURANT
! LHS is not an array section, no subscript triplets or vector subscripts.
PARIS%STREET(10,20) = S
! None of the parts are array sections, but the entire construct
!   is a section because STREET has a nonzero rank and is not
!   the rightmost part.
PARIS%STREET%ADDRESS(100) = BUILDING_T(.TRUE.)

! STREET(50:100,10) is an array section, making the LHS an array section
!   with rank=1, shape=(/51/).
```

```
            ! ADDRESS(123) must not be an array section because only one can appear
            !   in a reference to a structure component.
            PARIS%STREET(50:100,10)%ADDRESS(123)%RESIDENTIAL = .TRUE.
            END
```

Rank and shape of array sections

For an array section that is not a subobject of a structure component, the rank is
the number of subscript triplets and vector subscripts in the *section_subscript_list*.
The number of elements in the shape array is the same as the number of subscript
triplets and vector subscripts, and each element in the shape array is the number
of integer values in the sequence designated by the corresponding subscript triplet
or vector subscript.

For an array section that is a subobject of a structure component, the rank and
shape are the same as those of the part of the component that is an array name or
array section.

```
DIMENSION :: ARR1(10,20,100)
TYPE STRUCT2_T
  LOGICAL SCALAR_COMPONENT
END TYPE
TYPE STRUCT_T
  TYPE (STRUCT2_T), DIMENSION(10,20,100) :: SECTION
END TYPE

TYPE (STRUCT_T) STRUCT

! One triplet + one vector subscript, rank = 2.
! Triplet designates an extent of 10, vector subscript designates
!   an extent of 3, thus shape = (/ 10,3 /).
ARR1(:, (/ 1,3,4 /), 10) = 0

! One triplet, rank = 1.
! Triplet designates 5 values, thus shape = (/ 5 /).
STRUCT%SECTION(1,10,1:5)%SCALAR_COMPONENT = .TRUE.

! Here SECTION is the part of the component that is an array,
!   so rank = 3 and shape = (/ 10,20,100 /), the same as SECTION.
STRUCT%SECTION%SCALAR_COMPONENT = .TRUE.
```

Array constructors

An array constructor is a sequence of specified scalar values. It constructs a
rank-one array whose element values are those specified in the sequence. You can
construct arrays of rank greater than one using an intrinsic function. See
"RESHAPE(SOURCE, SHAPE, PAD, ORDER)" on page 690 for details.

Syntax

```
►►─┬─(/ac_spec/)─┬──────────────────────────────────────────────►◄
   └─[ ac_spec ]─┘
```

where *ac_spec* is:

```
►►─┬─i_d_type_spec──::─────────────────────────────────────────►◄
   │                        ┌─,──────┐
   │                        │        │
   └─i_d_type_spec──::─┘    ▼─ac_value─┘
```
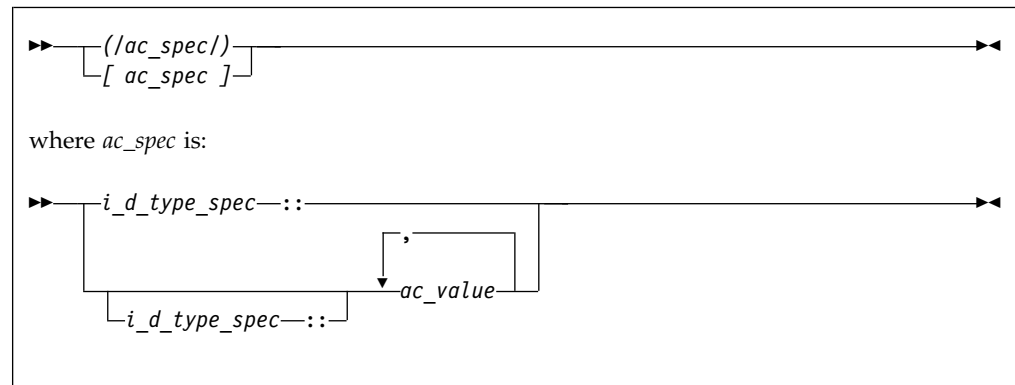
i_d_type_spec

>   Is an *intrinsic_type_spec* or *derived_type_spec*. For a list of possible type
>   specifications, see "Type Declaration" on page 476.
>
>   You cannot use **BYTE** as an *intrinsic_type_spec* in an array constructor.

ac_value
>   is an expression or implied-**DO** list that provides values for array elements.

Rules

An *i_d_type_spec* specifies the type and type parameters of the array constructor.
Each *ac_value* expression must be compatible with intrinsic assignment to a
variable with the type and type parameters. Each value is converted to the type
parameters of the array constructor.

*   If you specify an intrinsic type, each *ac_value* expression in the array constructor
    must be of an intrinsic type compatible with the type you specify.
*   If you specify a derived type, all *ac_value* expressions in the array constructor
    must be of that derived type and have the same kind type parameter values as
    the type you specify.

If *i_d_type_spec* is omitted, each *ac_value* expression in the array constructor must
have the same type and type parameters.

If *i_d_type_spec* appears without an *ac_value*, a zero-sized rank-one array is created.

The *ac_value* complies with the following rules:

*   If it is a scalar expression, its value specifies an element of the array constructor.
*   If it is an array expression, the values of the elements of the expression, in array
    element order, specify the corresponding sequence of elements of the array
    constructor.
*   If it is an implied-**DO** list, it is expanded to form an *ac_value* sequence under the
    control of the *implied_do_variable*, as in the **DO** construct.

F2003

If *ac_value* is a polymorphic entity, its declared type is used. Because unlimited
polymorphic entities have no declared type, you cannot use them for *ac_value*.

IBM If you compile your program with **-qxlf2003=dynamicacval**, the dynamic type

of *ac_value* is used, and you can use unlimited polymorphic entities for *ac_value*.
◀IBM For more details about using polymorphic entities for *ac_value*, see Example 2.

For more information about unlimited polymorphic entities, the declared and dynamic types of these entities, see Polymorphic entities.

◀F2003

Examples

Example 1: Different array constructors

```
INTEGER, DIMENSION(5) :: a, b, c, d(2, 2)
CHARACTER(5), DIMENSION(3) :: color

! Assign values to all elements in a
a = (/1, 2, 3, 4, 5/)

! Assign values to some elements
a(3:5) = (/0, 1, 0/)

! Construct temporary logical mask
c = MERGE(a, b, (/T, F, T, T, F/))

! The array constructor produces a rank-one array, which
! is turned into a 2x2 array that can be assigned to d.
d = RESHAPE(SOURCE = (/1, 2, 1, 2/), SHAPE = (/2, 2/))

! Here, the constructor linearizes the elements of d in
! array-element order into a one-dimensional result.
PRINT *, a((/d/))

! Without a type_spec,each character literal must be of length 5
color = ['RED  ', 'GREEN', 'BLUE ']

! With a type_spec, padding and truncation of each character literal occurs
color = [CHARACTER(5) :: 'RED', 'GREEN', 'BLUE']
```

F2003▶

Example 2: Polymorphic entities as *ac_value*

```
PROGRAM PolyAcValues
  TYPE base
    INTEGER :: i
  END TYPE

  TYPE, EXTENDS(base) :: child
    INTEGER :: j
  END TYPE

  TYPE(base) :: baseType = base(3)
  TYPE(child) :: childType = child(4, 6)

  ! Declare a polymorphic entity of base type
  CLASS(base), ALLOCATABLE :: baseClass
  ! Declare an unlimited polymorphic entity. It has no declared type.
  ! Its dynamic type can be any derived type or intrinsic type
  CLASS(*), ALLOCATABLE :: anyClass
  ! Declare a deferred-shape array of unlimited polymorphic entities
  CLASS(*), ALLOCATABLE :: anyClassArr(:)

  ! Allocate a child item to baseClass. The dynamic type of bassClass is child.
  ALLOCATE(baseClass, source = childType)
```

```
! Polymorphic entities used in the array constructor
ALLOCATE(anyClassArr(2), source = [baseClass, baseClass])

! Because the compiler uses the declared type, which is base, and the result
! is "Base item: 4 4". If you specify -qxlf2003=dynamicacval, the compiler uses
! the dynamic type, which is child, and the result is "Child item: 4,6 4,6".
CALL printAny(anyClassArr, 2)
DEALLOCATE(anyClassArr)
DEALLOCATE(baseClass)

! Allocate a base item to anyClass. The dynamic type of anyClass is base.
ALLOCATE(anyClass, source = baseType)
! Unlimited polymorphic entities used in the array constructor
ALLOCATE(anyClassArr(2), source = [anyClass, anyClass])

! If you specify -qxlf2003=dynamicacval, the use of unlimited polymorphic
! entities in the array constructor is valid, and the compiler uses the
! dynamic type, which is base. The result is "Base item: 3 3"; Otherwise,
! a severe error is issued at compile time.
CALL printAny(anyClassArr, 2)
DEALLOCATE(anyClassArr)
DEALLOCATE(anyClass)

CONTAINS
  SUBROUTINE printAny(printItem, len)
    CLASS(*) :: printItem(len)

    DO i = 1, len
      SELECT TYPE (item => printItem(i))
        TYPE IS (base)
          PRINT *, 'Base item: ', item
        TYPE IS (child)
          PRINT *, 'Child item: ', item
      END SELECT
    END DO
  END SUBROUTINE
END PROGRAM
```

F2003

Related information

- The -qxlf2003=dynamicacval option
- "Polymorphic entities (Fortran 2003)" on page 20

Implied-DO list for an array constructor

Implied-**DO** loops in array constructors help to create a regular or cyclic sequence of values, to avoid specifying each element individually.

A zero-sized array of rank one is formed if the sequence of values generated by the loop is empty.

```
►►──(──ac_value_list──,──implied_do_variable── = ──expr1──,──expr2──────────────)──────────────►◄
                                                                  └─,──expr3─┘
```
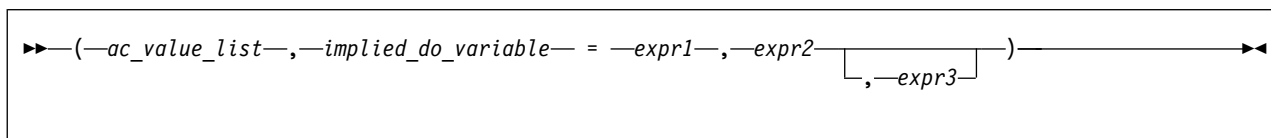
implied_do_variable

> is a named scalar integer IBM or real IBM variable.

> In a nonexecutable statement, the type must be integer. You must not reference the value of an *implied_do_variable* in the limit expressions *expr1*

or *expr2*. Loop processing follows the same rules as for an implied-**DO** in
"DATA" on page 327, and uses integer or real arithmetic depending on the
type of the implied-**DO** variable.

The variable has the scope of the implied-**DO**, and it must not have the
same name as another implied-**DO** variable in a containing array
constructor implied-**DO**:

```
M = 0
PRINT *, (/ (M, M=1, 10) /) ! Array constructor implied-DO
PRINT *, M                  ! M still 0 afterwards
PRINT *, (M, M=1, 10)       ! Non-array-constructor implied-DO
PRINT *, M                  ! This one goes to 11
PRINT *, (/ ((M, M=1, 5), N=1, 3) /)
! The result is a 15-element, one-dimensional array.
! The inner loop cannot use N as its variable.
```

expr1, expr2, **and** *expr3*

        are scalar integer IBM or real IBM expressions

```
PRINT *, (/ (I, I = 1, 3) /)
! Sequence is (1, 2, 3)
PRINT *, (/ (I, I = 1, 10, 2) /)
! Sequence is (1, 3, 5, 7, 9)
PRINT *, (/ (I, I+1, I+2, I = 1, 3) /)
! Sequence is (1, 2, 3, 2, 3, 4, 3, 4, 5)

PRINT *, (/ ( (I, I = 1, 3), J = 1, 3 ) /)
! Sequence is (1, 2, 3, 1, 2, 3, 1, 2, 3)

PRINT *, (/ ( (I, I = 1, J), J = 1, 3 ) /)
! Sequence is (1, 1, 2, 1, 2, 3)

PRINT *, (/2,3,(I, I+1, I = 5, 8)/)
! Sequence is (2, 3, 5, 6, 6, 7, 7, 8, 8, 9).
! The values in the implied-DO loop before
!   I=5 are calculated for each iteration of the loop.
```

Contiguity (Fortran 2008)

Contiguous objects occupy a contiguous block of memory. The use of contiguous
objects makes it easier to enable optimizations that depend on the memory layout
of the objects.

An object is contiguous if it meets one of the following requirements:
- It is an object that has the **CONTIGUOUS** attribute.
- It is a whole array that is neither an array pointer nor an assumed-shape array.
- It is an array that is allocated by an **ALLOCATE** statement.
- It is a pointer that is associated with a contiguous target.
- It is an assumed-shape array that is argument associated with a contiguous
  array.
- It is a nonzero-sized array section that meets all the following requirements:
  - Its base object is contiguous.
  - It does not have a *vector_subscript*.
  - The elements of the section, in array element order, are a subset of the base
    object elements that are consecutive in array element order.
  - If the array is of type character and a *substring_range* is used, the
    substring_range specifies all the characters of the parent string.
  - Only its rightmost *comp_name* has nonzero rank.

– It is not the real or imaginary part of an array of type complex.

An object is not contiguous if it is an array subobject and meets all the following requirements:
• The object has two or more elements.
• The elements of the object in array element order are not consecutive in the elements of the base object.
• The object is not of type character that has zero length.
• The object is not of a derived type that only contains zero-sized arrays and characters that have zero length.

Note: In addition to the preceding scenarios, XL Fortran determines whether the object is contiguous, based on its own rules.

Simply contiguous

A simply contiguous array is one that the XL Fortran compiler can determine to be contiguous at compile time.

A *section_subscript_list* specifies a simply contiguous array section only if it meets all the following requirements:
• It does not have a *vector_subscript*.
• All but the last *subscript_triplet* is a colon.
• The last *subscript_triplet* does not have a *stride*.
• No *subscript_triplet* is preceded by a *section_subscript* that is a *subscript*.

An array subobject designator is simply contiguous only if it meets one of the following requirements:
• An *object_name* that has the **CONTIGUOUS** attribute
• An *object_name* that is not a pointer, not an assumed shape array, `TS 29113` and not a nonallocatable assumed-rank object `TS 29113`
• A structure component whose rightmost *part_name* is an array. The rightmost *part_name* either has the **CONTIGUOUS** attribute or is not a pointer.
• An array section that meets all the following requirements:
  – It is not a *complex part designator*.
  – It does not have a *substring_range*.
  – Its rightmost *comp_name* has nonzero rank.
  – Its rightmost *part_name* has the **CONTIGUOUS** attribute or is neither of assumed shape nor a pointer.
  – It either does not have a *section_subscript_list*, or has a *section_subscript_list* which specifies a simply contiguous section.

An array variable is simply contiguous only if it meets one of the following requirements:
• It is an array subobject designator that is simply contiguous.
• It is a reference to a function that returns a pointer with the **CONTIGUOUS** attribute.

Note: In addition to the preceding scenarios, XL Fortran may determine contiguity at compile time, based on its own rules.

Related information
- Array sections
- CONTIGUOUS

Expressions involving arrays

Arrays can be used in the same kinds of expressions and operations as scalars. Intrinsic operations, assignments, or elemental procedures can be applied to one or more arrays.

For intrinsic operations, in expressions involving two or more array operands, the arrays must have the same shape so that the corresponding elements of each array can be assigned to or be evaluated. In a defined operation arrays can have different shapes. Arrays with the same shape are *conformable*. In a context where a conformable entity is expected, you can also use a scalar value: it is conformable with any array, such that it is treated like an array where each array element has the value of the scalar.

Examples

```
INTEGER, DIMENSION(5,5) :: A,B,C
REAL, DIMENSION(10) :: X,Y
! Here are some operations on arrays
A = B + C          ! Add corresponding elements of both arrays.
A = -B             ! Assign the negative of each element of B.
A = MAX(A,B,C)     ! A(i,j) = MAX( A(i,j), B(i,j), C(i,j) )
X = SIN(Y)         ! Calculate the sine of each element.
! These operations show how scalars are conformable with arrays
A = A + 5          ! Add 5 to each element.
A = 10             ! Assign 10 to each element.
A = MAX(B, C, 5)   ! A(i,j) = MAX( B(i,j), C(i,j), 5 )

END
```

Related information

"Elemental intrinsic procedures" on page 567
"Intrinsic assignment" on page 117
"WHERE" on page 493 shows a way to assign values to some elements in an array but not to others
"FORALL construct" on page 127

Chapter 6. Expressions and assignment

Introduction to expressions and assignment

An expression is a data reference or a computation, and is formed from operands, operators, and parentheses. An expression, when evaluated, produces a value, which has a declared type, a dynamic type, a shape, and possibly type parameters.

An *operand* is either a scalar or an array. An *operator* is either intrinsic or defined. A unary operation has the form:

- *operator operand*

A binary operation has the form:

- *operand₁ operator operand₂*

Any expression contained in parentheses is treated as a data entity. Parentheses can be used to specify an explicit interpretation of an expression. They can also be used to restrict the alternative forms of the expression, which can help control the magnitude and accuracy of intermediate values during evaluation of the expression. For example, the two expressions

```
(I*J)/K
I*(J/K)
```

are mathematically equivalent, but may produce different computational values as a result of evaluation.

Primary

A *primary* is the simplest form of an expression. It can be one of the following elements:

- A type parameter inquiry such as a%kind
- A type parameter name
- A data object
- An array constructor
- A structure constructor
- IBM A complex constructor IBM
- A function reference
- An expression enclosed in parentheses

A primary that is a data object must not be an assumed-size array.

Examples of primaries

```
12.3            ! Constant
'ABCDEFG'(2:3)  ! Subobject of a constant
VAR             ! Variable name
(/7.0,8.0/)     ! Array constructor
EMP(6,'SMITH')  ! Structure constructor
SIN(X)          ! Function reference
(T-1)           ! Expression in parentheses
```

Type, parameters, and shape

The type, type parameters, and shape of a primary are determined as follows:

- A data object or function reference acquires the type, type parameters, and shape of the object or function reference, respectively. The type, parameters, and shape of a generic function reference are determined by the type, parameters, and ranks of its actual arguments.
- A type parameter inquiry or type parameter name is a scalar integer with the kind of the type parameter.
- A structure constructor is a scalar and its type and parameters are determined by the *derived_type_spec* of the structure constructor.
- An array constructor has a shape determined by the number of constructor expressions, and its type and parameters are determined by those of the constructor expressions.
- A parenthesized expression acquires the type, parameters, and shape of the expression.

If a pointer appears as a primary in an operation in which it is associated with a nonpointer dummy argument, the target is referenced. The type, parameters, and shape of the primary are those of the target. If the pointer is not associated with a target, it can appear only as an actual argument in a procedure reference whose corresponding dummy argument is a pointer, or as the target in a pointer assignment statement. A disassociated pointer can also appear as an actual argument to the ASSOCIATED intrinsic inquiry function.

Given the intrinsic operation *[expr1] op expr2*, the shape of the operation is the shape of *expr2* if *op* is unary or if *expr1* is a scalar. Otherwise, its shape is that of *expr1*.

The type and shape of an expression are determined by the operators and by the types and shapes of the expression's primaries. The type of the expression can be intrinsic or derived. An expression of intrinsic type has a kind parameter and, if it is of type character, it also has a length parameter. An expression of derived type can have both kind and length parameters.

Constant expressions

A constant expression is an expression in which each operation is intrinsic and each primary is one of the following items:

F2003 **Note:** In Fortran 2003, the constant expression is known as initialization expression. F2003
- A constant or a subobject of a constant
- A constant expression that is enclosed in parentheses
- A DATA do_variable within a DATA implied_DO
- A previously declared kind type parameter of the type being defined or of its parent type within the derived type definition
- An array constructor where each element and the bounds and stride of each implied-DO are constant expressions
- A structure constructor where each component expression meets one of the following requirements:

Table 15. Requirements for structure constructors being constant expressions

If the component expression corresponds to ...	It must be ...
An allocatable component	A reference to the intrinsic function **NULL**

Table 15. Requirements for structure constructors being constant expressions (continued)

If the component expression corresponds to ...	It must be ...
A pointer component	Either an initialization target or a reference to the intrinsic function **NULL**
Any other component	A constant expression

- A reference to an elemental intrinsic function where each argument is a constant expression
- A reference to the vector intrinsic function VEC_CONVERT where each argument is a constant expression
- A reference to the intrinsic function **NULL** where the argument must not have assumed type parameters and the type parameters must be defined by constant expressions
- A reference to the intrinsic function **TRANSFER** where each argument is a constant expression and each ultimate pointer component of the **SOURCE** argument is disassociated
- A reference to the transformational intrinsic function IEEE_SELECTED_REAL_KIND from the intrinsic module IEEE_ARITHMETIC, where each argument is a constant expression
- A reference to a transformational intrinsic function other than **COMMAND_ARGUMENT_COUNT**, **NULL**, and **TRANSFER**, where each argument is a constant expression
- A specification inquiry where each designator or function argument is either a constant expression or a variable with properties that are not assumed, deferred, or defined by expressions that are not constant expressions

and where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is a constant expression.

If a constant expression includes a specification inquiry that depends on a **TYPE** parameter or an array bound of an object specified in the same specification part, the type parameter or array bound must be specified in a prior specification of the specification part. The prior specification can be to the left of the specification inquiry in the same statement, but must not be within the same entity declaration.

Examples

Examples of different constant expressions:

```
-48.9
name('Pat','Doe')
TRIM('ABC    ')
(MOD(9,4)**3.5)
3.4**3
KIND(57438)
(/'desk','lamp'/)
'ab'//'cd'//'ef'
```

Example of an elemental intrinsic function, SIN, used in a constant expression:

```
integer, parameter :: foo = 42.0 * sin(0.5)
```

Example of a transformational intrinsic function, CSHIFT, used in a constant expression:

```
integer, parameter :: a(3) = (/ 1, 2, 3 /)
integer, parameter :: a_cshifted(3) = cshift(a,2)
```

Specification expressions

A specification expression is an expression with limitations that you can use to specify items such as character lengths and array bounds.

A specification expression is a scalar, integer, restricted expression.

A *restricted expression* is an expression in which each operation is intrinsic and each primary is:
- A type parameter of the derived type being defined.
- A constant or a subobject of a constant.
- A variable that is a dummy argument that has neither the **OPTIONAL** nor the **INTENT(OUT)** attribute, or a subobject of such a variable.
- A variable that is in a common block, or a subobject of such a variable.
- A variable accessible by use association or host association, or a subobject of such a variable.
- An array constructor where each element and the bounds and strides of each implied-**DO** are expressions whose primaries are either restricted expressions or implied-**DO** variables.
- A structure constructor where each component is a restricted expression.
- A specification inquiry where each designator or function argument is either a restricted expression or a variable with properties that are not assumed, deferred, or defined by an expression that is not a restricted expression.
- A reference to any remaining intrinsic functions defined in this document where each argument is a restricted expression.
- IBM A reference to a system inquiry function, where any arguments are restricted expressions. IBM
- Any subscript or substring expression must be a restricted expression.
- A reference to a specification function, where any arguments are restricted expressions.

A *specification inquiry* is a reference to:
- An intrinsic inquiry function
- A type parameter inquiry
- An IEEE inquiry function

You can use a *specification function* in a specification expression. A function is a specification function if it is a pure function that is not an intrinsic, internal or statement function. A specification function cannot have a dummy procedure argument.

A variable in a specification expression must have its type and type parameters, if any, specified by a previous declaration in the same scoping unit, or by the implicit typing rules in effect for the scoping unit, or by host or use association. If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

If a specification expression includes a reference to an inquiry function for a type parameter or an array bound of an entity specified in the same specification part, the type parameter or array bound must be specified in a prior specification of the specification part. If a specification expression includes a reference to the value of an element of an array specified in the same specification part, the array bounds must be specified in a prior declaration. The prior specification can be to the left of the inquiry function in the same statement.

Examples

```
LBOUND(C,2)+6    ! C is an assumed-shape dummy array
ABS(I)*J         ! I and J are scalar integer variables
276/NN(4)        ! NN is accessible through host association
```

The following example shows how a user-defined pure function, fact, can be used in the specification expression of an array-valued function result variable:

```
MODULE MOD
CONTAINS
  INTEGER PURE FUNCTION FACT(N)
  INTEGER, INTENT(IN) :: N
  ...
  END FUNCTION FACT
END MODULE MOD

PROGRAM P
  PRINT *, PERMUTE('ABCD')
  CONTAINS
  FUNCTION PERMUTE(ARG)
    USE MOD
    CHARACTER(*), INTENT(IN) :: ARG
    ...
    CHARACTER(LEN(ARG)) :: PERMUTE(FACT(LEN(ARG)))
    ...
  END FUNCTION PERMUTE
END PROGRAM P
```

Operators and expressions

This section contains details on the XL Fortran expressions listed in the *XL Fortran Expressions* table. For information on the order of evaluation precedence see, How expressions are evaluated.

Table 16. XL Fortran expressions

Arithmetic	Logical
Character	Primary
General	Relational

Arithmetic

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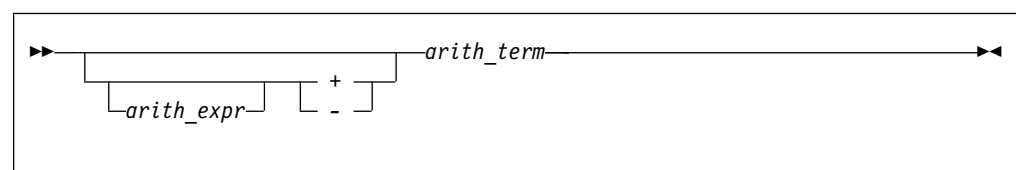
The form of *arith_term* is:

```
►►─┬──────────────────────┬──arith_factor──────────────────────────────►◄
   │ ┌──arith_term──┬─ / ─┐│
   └─┘              ├─ * ─┘│
                    
```
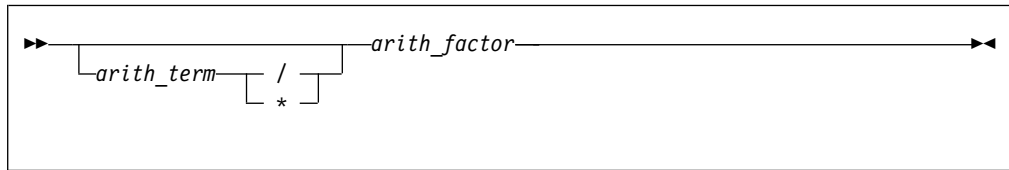
The form of *arith_factor* is:

```
►►──arith_primary─┬────────────────────────┬───────────────────────────►◄
                  └─ ** ─arith_factor──────┘
```
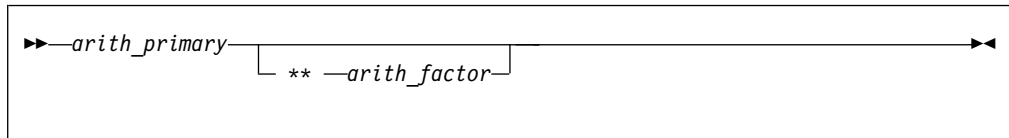
An *arith_primary* is a primary of arithmetic type.

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

Arithmetic Operator	Representation	Precedence
**	Exponentiation	First
*	Multiplication	Second
/	Division	Second
+	Addition or identity	Third
-	Subtraction or negation	Third

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 evaluated as (2+3)+4, although a processor can interpret the expression in another way if it is mathematically equivalent and respects any parentheses.

The factors are evaluated from left to right when evaluating a term containing two or more multiplication or division operators. For example, 2*3*4 is evaluated as (2*3)*4.

The primaries are combined from right to left when evaluating a factor containing two or more exponentiation operators. For example, 2**3**4 is evaluated as 2**(3**4). (Again, mathematical equivalents are allowed.)

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, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. Thus, -A**3 is evaluated as -(A**3).

Note that expressions containing two consecutive arithmetic operators, such as A**-B or A*-B, are not allowed. You can use expressions such as A**(-B) and A*(-B).

If an expression specifies the division of an integer by an integer, the result is rounded to an integer closer to zero. For example, `(-7)/3` has the value `-2`.

IBM For details of exception conditions that can arise during evaluation of floating-point expressions, see Detecting and trapping floating-point exceptions. **IBM**

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

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 on a pair of operands.

Notation: *T(param)*, where *T* is the data type (I: integer, R: real, X: complex) and *param* is the kind type parameter.

Table 17. Result types for binary arithmetic operators

first operand		second operand								
	I(1)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(1)	I(1)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(2)	I(2)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(4)	I(4)	I(4)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(8)	I(8)	I(8)	I(8)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(4)	R(4)	R(4)	R(4)	R(4)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(16)	X(8)	X(8)	X(16)
R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	X(16)	X(16)	X(16)
X(4)	X(4)	X(4)	X(4)	X(4)	X(4)	X(8)	X(16)	X(4)	X(8)	X(16)
X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(16)	X(8)	X(8)	X(16)
X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)

Note: IBM

1. XL Fortran implements integer operations using **INTEGER(4)** arithmetic, or **INTEGER(8)** arithmetic if data items are 8 bytes in length. If the intermediate result is used in a context requiring **INTEGER(1)** or **INTEGER(2)** data type, it is converted as required.

```
INTEGER(2) I2_1, I2_2, I2_RESULT
INTEGER(4) I4
I2_1 = 32767          ! Maximum I(2)
I2_2 = 32767          ! Maximum I(2)
I4 = I2_1 + I2_2
PRINT *, "I4=", I4    ! Prints I4=-2

I2_RESULT = I2_1 + I2_2  ! Assignment to I(2) variable
I4 = I2_RESULT           ! and then assigned to an I(4)
PRINT *, "I4=", I4       ! Prints I4=-2
END
```

Character

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

```
►►─┬─────────────────┬──char_primary──────────────────────────────►◄
    └─char_expr──//─┘
```
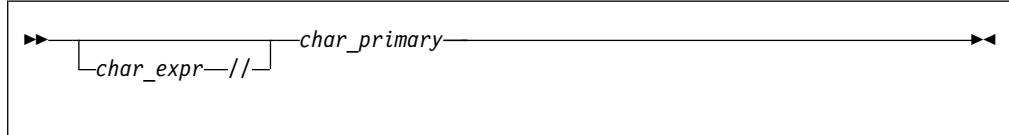
char_primary is a primary of type character. All character primaries in the expression must have the same kind type parameter, which is also the kind type parameter of the result.

The only character operator is //, representing concatenation.

In a character expression containing one or more concatenation operators, the primaries are joined to form one string whose length is equal to the sum of the lengths of the individual primaries. For example, `'AB'//'CD'//'EF'` evaluates to `'ABCDEF'`, a string 6 characters in length.

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

A character expression can include concatenation of an operand when you declare the length with an asterisk in parentheses. This indicates inherited length. In this case, the actual length depends on whether you use the inherited length character string to declare:

- A dummy argument specified in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement. The length of the dummy argument assumes the length of the associated actual argument on invocation.
- A named constant. The character expression takes on the length of the constant value.
- The length of an external function result. The calling scoping unit must not declare the function name with an asterisk. On invocation, the length of the function result assumes this defined length.

Examples

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

General

The general form of an expression (*general_expr*) is:

```
►►─┬──────────────────────────────────┬──expr──────────────────────►◄
    └─general_expr──defined_binary_op─┘
```
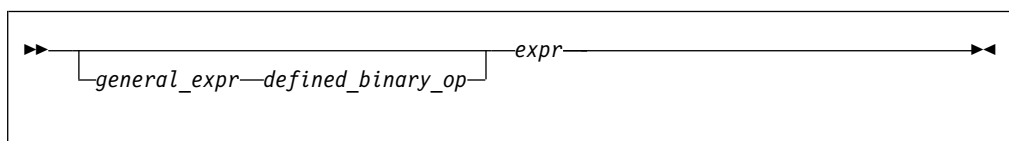
defined_binary_op
> is a defined binary operator. See "Extended intrinsic and defined operations" on page 113.

expr   is one of the kinds of expressions defined below.

There are four kinds of intrinsic expressions: arithmetic, character, relational, and logical.

Logical

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

```
>>──┬──────────────────────────────────┬──logical_disjunct──────────><
    └─logical_expr──┬─.EQV.──┬──────────┘
                    ├─.NEQV.─┤
                    │   (1)  │
                    └─.XOR.──┘

Notes:
1   XL Fortran logical operator
```
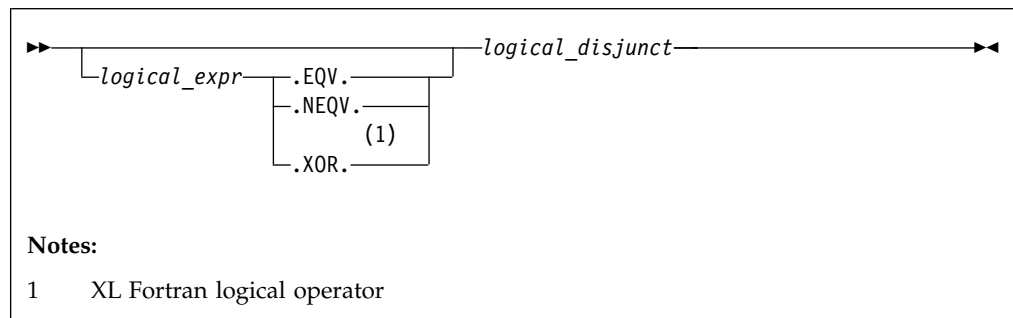
The form of a *logical_disjunct* is:

```
>>──┬──────────────────────┬──logical_term──────────────────────────><
    └─logical_disjunct─.OR.─┘
```
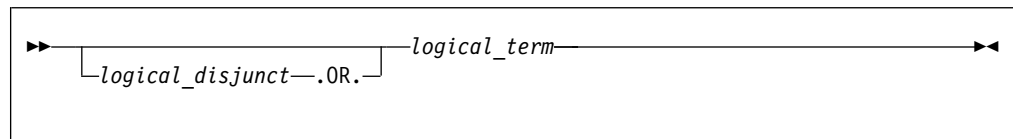
The form of a *logical_term* is:

```
>>──┬────────────────────┬──logical_factor──────────────────────────><
    └─logical_term─.AND.──┘
```
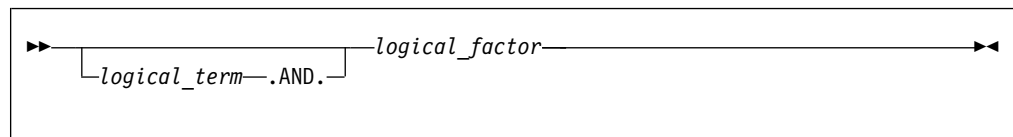
The form of a *logical_factor* is:

```
>>──┬───────┬──┬─logical_primary─┬───────────────────────────────────><
    └─.NOT.─┘  └─rel_expr─────────┘
```
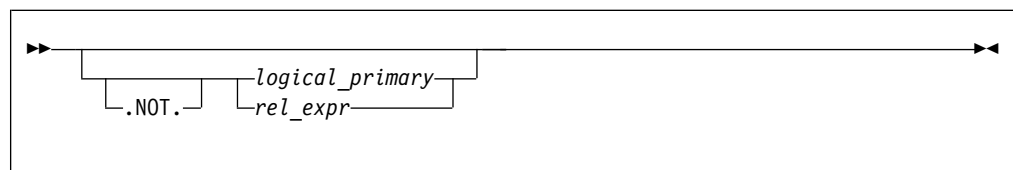
logical_primary is a primary of type logical.

rel_expr is a relational expression.

The logical operators are:

Logical Operator	Representing	Precedence
.NOT.	Logical negation	First (highest)
.AND.	Logical conjunction	Second
.OR.	Logical inclusive disjunction	Third
.XOR. **1**	Logical exclusive disjunction	Fourth (lowest)
.EQV.	Logical equivalence	Fourth (lowest)
.NEQV.	Logical nonequivalence	Fourth (lowest)

Note:

1. XL Fortran logical operator.

IBM The **.XOR.** operator is treated as an intrinsic operator only when the **-qxlf77=intxor** compiler option is specified. (See the **-qxlf77** option in the *XL Fortran Compiler Reference* for details.) Otherwise, it is treated as a defined operator. If it is treated as an intrinsic operator, it can also be extended by a generic interface. **IBM**

The precedence of the operators determines the order of evaluation when a logical expression containing two or more operators having different precedences is evaluated. 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

Given that *x1* and *x2* represent logical values, use the following tables to determine the values of logical expressions:

x1	.NOT. x1
True	False
False	True

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 a logical expression does not need to be completely evaluated 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, the evaluation of the function reference is not required to determine that this expression is true.

XL Fortran evaluates a logical expression to a **LOGICAL(n)** or **INTEGER(n)** result, where n is the kind type parameter. The value of n depends on the kind parameter of each operand.

By default, for the unary logical operator **.NOT.**, n will be the same as the kind type parameter of the operand. For example, if the operand is **LOGICAL(2)**, the result will also be **LOGICAL(2)**.

The following table shows the resultant type for unary operations:

OPERAND	RESULT of Unary Operation
BYTE **1**	INTEGER(1) **1**
LOGICAL(1)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)
Typeless **1**	Default integer **1**

Note:

1. IBM extension

If the operands are of the same length, n will be that length.

IBM For binary logical operations with operands that have different kind type parameters, the kind type parameter of the expression is the same as the larger length of the two operands. For example, if one operand is **LOGICAL(4)** and the other **LOGICAL(2)**, the result will be **LOGICAL(4)**. **IBM**

The following table shows the resultant type for binary operations:

Table 18. Result Types for binary logical expressions

	second operand					
first operand	*BYTE	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	*Typeless
*BYTE	*INTEGER(1)	*LOGICAL(1)	*LOGICAL(2)	*LOGICAL(4)	*LOGICAL(8)	*INTEGER(1)
LOGICAL(1)	LOGICAL(1)	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(8)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)
*Typeless	*INTEGER(1)	*LOGICAL(1)	*LOGICAL(2)	*LOGICAL(4)	*LOGICAL(8)	*Default Integer

Note: * Resultant types for binary logical expressions in XL Fortran

If the expression result is to be treated as a default integer but the value cannot be represented within the value range for a default integer, the constant is promoted to a representable kind.

Primary

The form of a primary expression is:

```
►►──┬─────────────────────┬──primary──────────────────────────────►◄
     └─defined_unary_op─┘
```
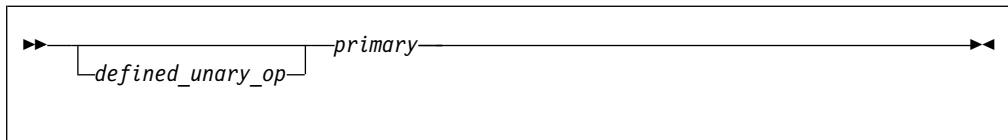
defined_unary_op
> is a defined unary operator. See "Extended intrinsic and defined operations" on page 113.

Relational

A relational expression (*rel_expr*), when evaluated, produces a result of type logical, and can appear wherever a logical expression can appear. It 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──relational_operator──arith_expr2──────────────────►◄
```
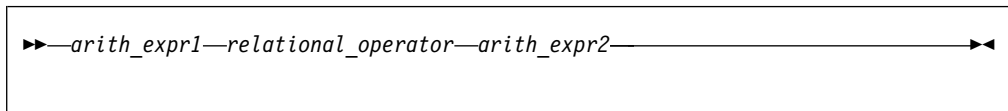
arith_expr1 **and arith_expr2**
> are each an arithmetic expression. Complex expressions can only be specified if *relational_operator* is **.EQ.**, **.NE.**, <>, ==, or /=.

relational_operator
> is any of:

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

An arithmetic relational expression is interpreted 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 the types or kind type parameters of the expressions differ, their values are converted to the type and kind type parameter of the expression (`arith_expr1 + arith_expr2`) before evaluation.

Examples

```
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──relational_operator──char_expr2────────────────────►◄
```

char_expr1 **and** *char_expr2*
>   are each character expressions

relational_operator
>   is any of the relational operators described in "Arithmetic relational expressions" on page 112.

For all relational operators, the collating sequence is used to interpret a character relational expression. The character expression whose value is lower in the collating sequence is less than the other expression. The character expressions are evaluated one character at a time from left to right. You can also use the intrinsic functions (**LGE**, **LLT**, and **LLT**) to compare character strings in the order specified by the ASCII collating sequence. For all relational operators, if the operands are of unequal length, the shorter is extended on the right with blanks. If both *char_expr1* and *char_expr2* are of zero length, they are evaluated as equal.

IBM Even if *char_expr1* and *char_expr2* are multibyte characters (MBCS) in XL Fortran, the ASCII collating sequence is still used. **IBM**

Examples

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

Extended intrinsic and defined operations

A defined operation is either a defined unary operation or a defined binary operation. It is defined by a function and a generic interface (see "Interface blocks" on page 164 and "Type-bound procedures (Fortran 2003)" on page 60). A defined operation is not an intrinsic operation, although an intrinsic operator can be extended in a defined operation. For example, to add two objects of derived type, you can extend the meaning of the intrinsic binary operator for addition (+). If an extended intrinsic operator has typeless operands, the operation is evaluated intrinsically.

The operand of a unary intrinsic operation that is extended must not have a type that is required by the intrinsic operator. Either or both of the operands of a binary intrinsic operator that is extended must not have the types or ranks that are required by the intrinsic operator.

The defined operator of a defined operation must be defined in a generic interface.

A defined operator is an extended intrinsic operator or has the form:

Notes:

1   XL Fortran defined operator

2   XL Fortran defined operator
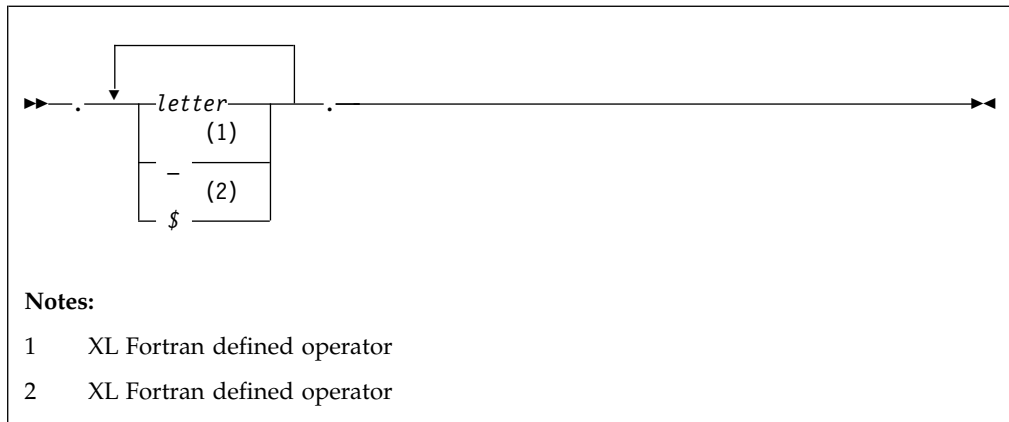
In Fortran 90 and Fortran 95, a defined operator must not contain more than 31 letters, and must not be the same as any intrinsic operator or logical literal constant. In Fortran 2003 the letter limit for a defined operator is 63.

See "Generic interface blocks" on page 167 for details on defining and extending operators in an interface block. See "Type-bound procedures (Fortran 2003)" on page 60 for details on defining and extending operators that are bound to a derived type.

How expressions are evaluated

Precedence of operators

An expression can contain more than one kind of operator. When it does, the expression is evaluated from left to right, according to the following precedence among operators:

1. Defined unary
2. Arithmetic
3. Character
4. Relational
5. Logical
6. Defined binary

For example, the logical expression:

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

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

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

An extended intrinsic operator maintains its precedence. That is, the operator does not have the precedence of a defined unary operator or a defined binary operator.

Summary of interpretation rules

Primaries that contain operators are combined in the following order:

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

Arithmetic, character, relational, and logical expressions are evaluated according to the following rules:

- A variable or function must be defined at the time it is used. You must define an integer operand with an integer value, not a statement label value. All referenced characters in a character data object or referenced array elements in an array or array section must be defined at the time the reference is made. All components of a structure must be defined when a structure is referenced. A pointer must be associated with a defined target.

  Execution of an array element reference, array section reference, and substring reference requires the evaluation of its subscript, section subscript and substring expressions. Evaluation of any array element subscript, section subscript, substring expression, or the bounds and stride of any array constructor implied-**DO** does not affect, nor is it affected by, the type of the containing expression. See "Expressions involving arrays" on page 100. You cannot use any constant integer operation or floating-point operation whose result is not mathematically defined in an executable program. If such expressions are nonconstant and are executed, they are detected at run time. (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 of type real to a real power.

- The invocation of a function in a statement must not affect, or be affected by, the evaluation of any other entity within the statement in which the function reference appears. When the value of an expression is true, invocation of a function reference in the expression of a logical **IF** statement or a **WHERE** statement can affect entities in the statement that is executed. If a function reference causes definition or undefinition 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) = FUNC1(I)
Y = FUNC2(X) + X
```

  if the reference to FUNC1 defines I or the reference to FUNC2 defines X.

  The data type of an expression in which a function reference appears does not affect, nor is it affected by, the evaluation of the actual arguments of the function.

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

IBM Several compiler options affect the data type of the final result:

- When you use the **-qintlog** compiler option, you can mix integer and logical values in expressions and statements. The data type and kind type parameter of the result depends on the operands and the operator involved. In general:
  - For unary logical operators (**.NOT.**) and arithmetic unary operators (**+,-**):

Data Type of OPERAND	Data Type of RESULT of Unary Operation
BYTE	INTEGER(1)
INTEGER(n)	INTEGER(n)
LOGICAL(n)	LOGICAL(n)
Typeless	Default integer

where n represents the kind type parameter. n must not be replaced with a logical constant even if **-qintlog** is on, nor by a character constant even if **-qctyplss** is on, nor can it be a typeless constant. In the case of **INTEGER** and **LOGICAL** data types, the length of the result is the same as the kind type parameter of the operand.

– For binary logical operators (**.AND.**, **.OR.**, **.XOR.**, **.EQV.**, **.NEQV.**) and arithmetic binary operators (******, *****, **/**, **+**, **-**), the following table summarizes what data type the result has:

	second operand			
first operand	**BYTE**	**INTEGER(y)**	**LOGICAL(y)**	**Typeless**
BYTE	INTEGER(1)	INTEGER(y)	LOGICAL(y)	INTEGER(1)
INTEGER(x)	INTEGER(x)	INTEGER(z)	INTEGER(z)	INTEGER(x)
LOGICAL(x)	LOGICAL(x)	INTEGER(z)	LOGICAL(z)	LOGICAL(x)
Typeless	INTEGER(1)	INTEGER(y)	LOGICAL(y)	Default integer

Note: **z** is the kind type parameter of the result such that **z** is equal to the greater of **x** and **y**. For example, a logical expression with a **LOGICAL(4)** operand and an **INTEGER(2)** operand has a result of **INTEGER(4)**.

For binary logical operators (**.AND.**, **.OR.**, **.XOR.**, **.EQV.**, **.NEQV.**), the result of a logical operation between an integer operand and a logical operand or between two integer operands will be integer. The kind type parameter of the result will be the same as the larger kind parameter of the two operands. If the operands have the same kind parameter, the result has the same kind parameter.

• When you use the **-qlog4** compiler option and the default integer size is **INTEGER(4)**, logical results of logical operations will have type **LOGICAL(4)**, instead of **LOGICAL(n)** as specified in the table above. If you specify the **-qlog4** option and the default integer size is not **INTEGER(4)**, the results will be as specified in the table above.

• When you specify the **-qctyplss** option compiler option, XL Fortran treats character constant expressions as Hollerith constants. If one or both operands are character constant expressions, the data type and the length of the result are the same as if the character constant expressions were Hollerith constants. See the "Typeless" rows in the previous tables for the data type and length of the result.

See *Summary of compiler options by functional category* and *Detailed descriptions of the XL Fortran compiler options* in the *XL Fortran Compiler Reference* for information about compiler options. IBM

Using BYTE data objects (IBM extension)

Data objects of type **BYTE** can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** data object can be used.

The data types of **BYTE** data objects are determined by the context in which you use them. XL Fortran does not convert them before use. For example, the type of a named constant is determined by use, not by the initial value assigned to it.

- When you use a **BYTE** data object as an operand of an arithmetic, logical, or relational binary operator, the data object assumes:
  - An **INTEGER(1)** data type if the other operand is arithmetic, **BYTE**, or a typeless constant
  - A **LOGICAL(1)** data type if the other operand is logical
  - A **CHARACTER(1)** data type if the other operand is character
- When you use a **BYTE** data object as an operand of the concatenation operator, the data object assumes a **CHARACTER(1)** data type.
- When you use a **BYTE** data object as an actual argument to a procedure with an explicit interface, the data object assumes the type of the corresponding dummy argument:
  - **INTEGER(1)** for an **INTEGER(1)** dummy argument
  - **LOGICAL(1)** for a **LOGICAL(1)** dummy argument
  - **CHARACTER(1)** for a **CHARACTER(1)** dummy argument
- When you use a **BYTE** data object as an actual argument passed by reference to an external subprogram with an implicit interface, the data object assumes a length of 1 byte and no data type.
- When you use a **BYTE** data object as an actual argument passed by value (**VALUE** attribute), the data object assumes an **INTEGER(1)** data type.
- When you use a **BYTE** data object in a context that requires a specific data type, which is arithmetic, logical, or character, the data object assumes an **INTEGER(1)**, **LOGICAL(1)**, or **CHARACTER(1)** data type, respectively.
- A pointer of type **BYTE** cannot be associated with a target of type character, nor can a pointer of type character be associated with a target of type **BYTE**.
- When you use a **BYTE** data object in any other context, the data object assumes an **INTEGER(1)** data type.

Intrinsic assignment

Assignment statements are executable statements that define or redefine variables based on the result of expression evaluation.

A defined assignment is not intrinsic, and is defined by a subroutine and an interface. See "Defined assignment" on page 172.

The general form of an intrinsic assignment is:

```
▶▶──variable── = ──expression──────────────────────────────────────▶◀
```

F2008 If *variable* is polymorphic, the following rules apply:
- *variable* must be allocatable.
- *variable* must be type-compatible with *expression*, or the declared types of *variable* and *expression* must conform as specified in Table 19 on page 118. F2008

F2003 The shapes of *variable* and *expression* must conform unless *variable* is an allocatable array. If *variable* is an allocatable array, and **-qxlf2003=autorealloc** has been specified, then *variable* and *expression* must not be arrays of different ranks. **F2003**

variable must be an array if *expression* is an array (see "Expressions involving arrays" on page 100). If *expression* is a scalar and *variable* is an array, *expression* is treated as an array of the same shape as *variable*, with every array element having the same value as the scalar value of *expression*. *variable* must not be a many-one array section (see "Vector subscripts" on page 91 for details), and neither *variable* nor *expression* can be an assumed-size array.

The types of *variable* and *expression* must conform as specified in Table 19.

Table 19. Type conformance for the intrinsic assignment statement

Type of *variable*	**Type of** *expression*
Numeric	Numeric
Logical	Logical
Character	Character
Derived type	Derived type (same as *variable*)

In numeric assignment statements, *variable* and *expression* can specify different numeric types and different kind type parameters. For logical assignment statements, the kind type parameters can differ. For character assignment statements, the length type parameters can differ.

If the length of a character variable is greater than the length of a character expression, the character expression is extended on the right with blanks until the lengths are equal. If the length of the character variable is less than the character expression, the character expression is truncated on the right to match the length of the character variable.

If *variable* is a pointer, it must be associated with a definable target that has type, type parameters and shape that conform with those of *expression*. The value of *expression* is then assigned to the target associated with *variable*.

Both *variable* and *expression* can contain references to any portion of *variable*.

F2003 If *variable* is an allocated allocatable variable, it is deallocated if any of the following conditions are satisfied:

- *expression* is an array of different shape.
- Any of the corresponding length type parameter values of *variable* and *expression* differ.
- **F2008** *variable* is polymorphic and has a different dynamic type from that of *expression*. **F2008**

When *variable* is or becomes a deallocated allocatable variable, its allocation occurs as follows:

- If *expression* is an array, *variable* is allocated with the shape of *expression* and with each lower bound equal to the corresponding element of **LBOUND**(*expression*).
- If *variable* is an array and *expression* is scalar, *variable* is allocated with its original bounds.

- Each deferred type parameter of *variable* is equal to the corresponding type parameters of *expression*.
- `F2008` If *variable* is polymorphic, it is allocated with the same dynamic type as *expression*. `F2008`

Specify **-qxlf2003=autorealloc** for reallocation support. See the **-qxlf2003=autorealloc** option option in the *XL Fortran Compiler Reference* for more information. `F2003`

An assignment statement causes the evaluation of *expression* and all expressions within *variable* before assignment, the possible conversion of *expression* to the type and type parameters of *variable*, and the definition of *variable* with the resulting value. No value is assigned to *variable* if it is a zero-length character object or a zero-sized array.

A derived-type assignment statement is an intrinsic assignment statement if there is no accessible defined assignment for objects of this derived type. The derived type expression must be of the same declared type as the variable. Each kind type parameter of the variable must have the same value as the corresponding kind of expression. Each length type parameter of the variable must have the same value as the corresponding type parameter of expression unless the variable is allocatable, and its corresponding type parameter is deferred. See "Determining declared type for derived types" on page 70 for the rules that determine when two structures are of the same derived type. Assignment is performed as if each component of the expression is assigned to the corresponding component of the variable. Pointer assignment is executed for pointer components, defined assignment is performed for each nonpointer nonallocatable component of a type that has a defined assignment consistent with the component, and intrinsic assignment is performed for each other nonpointer nonallocatable component.

For intrinsic assignment of the objects of a derived type containing an allocatable component, the allocatable component of the variable receives the allocation status and, if allocated, the bounds and value of the corresponding component of the expression. This occurs as if the following sequence of steps applies:

1. If the component of *variable* is allocated, it is deallocated.
2. If the corresponding component of *expression* is allocated, the corresponding component of *variable* is allocated with the same type and type parameters as the component of *expression*. If it is an array, it is allocated with the same bounds.
3. The value of the component of *expression* is then assigned to the corresponding component of *variable* using:
   - Defined assignment if the declared type of the component has a defined assignment consistent with the component.
   - Intrinsic assignment for the dynamic type of that component otherwise.

When *variable* is a subobject, the assignment does not affect the definition status or value of other parts of the object.

Arithmetic conversion

For numeric intrinsic assignment, the value of *expression* may be converted to the type and kind type parameter of *variable*, as specified in the following table:

Type of *variable*	Value Assigned
Integer	INT(*expression*,KIND=KIND(*variable*))
Real	REAL(*expression*,KIND=KIND(*variable*))
Complex	CMPLX(*expression*,KIND=KIND(*variable*))

`IBM`

Note: Arithmetic integer operations for **INTEGER(8)** data items, including intermediate results, are performed using **INTEGER(8)** arithmetic. Arithmetic integer operations for **INTEGER(1)**, **INTEGER(2)**, and **INTEGER(4)** data objects, including intermediate results, are performed using **INTEGER(8)** arithmetic. If an intermediate result is used in a context requiring a smaller integer size, it is converted as required.

`IBM`

Character assignment

Only as much of the character expression as is necessary to define the character variable needs to be evaluated. For example:

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

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

BYTE assignment

`IBM` If *expression* is of an arithmetic type, arithmetic assignment is used. Similarly, if *expression* is of type character, character assignment is used, and if *expression* is of type logical, logical assignment is used. If the expression on the right is of type **BYTE**, arithmetic assignment is used. `IBM`

Examples

```
INTEGER I(10)
LOGICAL INSIDE
REAL R,RMIN,RMAX
REAL :: A=2.3,B=4.5,C=6.7
TYPE PERSON
   INTEGER(4) P_AGE
   CHARACTER(20) P_NAME
END TYPE
TYPE (PERSON) EMP1, EMP2
CHARACTER(10) :: CH = 'ABCDEFGHIJ'

I = 5                     ! All elements of I assigned value of 5

RMIN = 28.5 ; RMAX = 29.5
R = (-B + SQRT(B**2 - 4.0*A*C))/(2.0*A)
INSIDE = (R .GE. RMIN) .AND. (R .LE. RMAX)

CH(2:4) = CH(3:5)              ! CH is now 'ACDEEFGHIJ'

EMP1 = PERSON(45, 'Frank Jones')
EMP2 = EMP1

! EMP2%P_AGE is assigned EMP1%P_AGE using arithmetic assignment
! EMP2%P_NAME is assigned EMP1%P_NAME using character assignment

END
```

WHERE construct

The **WHERE** construct masks the evaluation of expressions and assignments of values in array assignment statements. It does this according to the value of a logical array expression.

```
►►─┬─WHERE_construct_statement─┬─────────────────────────────────►
   │   ┌────────────────────┐  │
   └─▼─where_body_construct─┴──┘

   ┌──────────────────────┐   ┌──────────────────┐
►──▼─masked_ELSEWHERE_block─┴──▼─ELSEWHERE_block─┴───────────────►
►─END_WHERE_statement──────────────────────────────────────────►◄
```
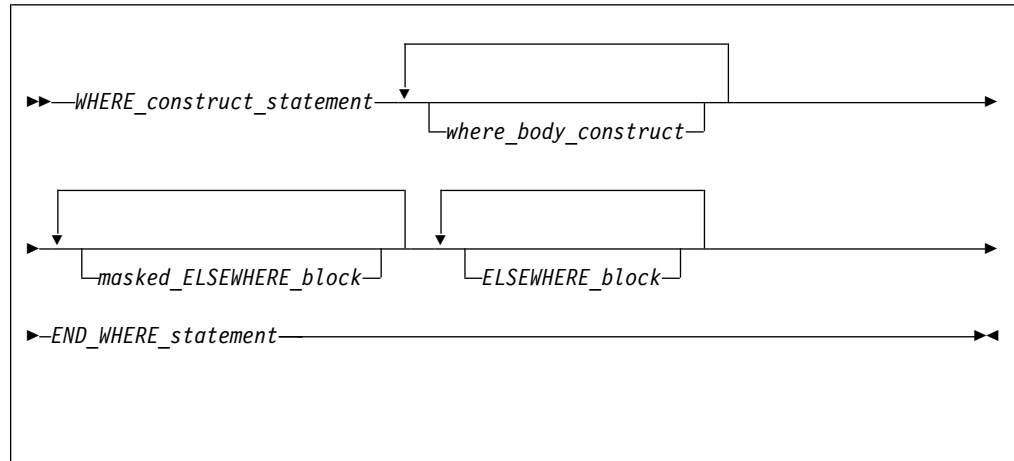
WHERE_construct_statement
   See "WHERE" on page 493 for syntax details.

where_body_construct

```
►►─┬─where_assignment_statement─┬───────────────────────────────►◄
   ├─WHERE_statement────────────┤
   └─WHERE_construct────────────┘
```
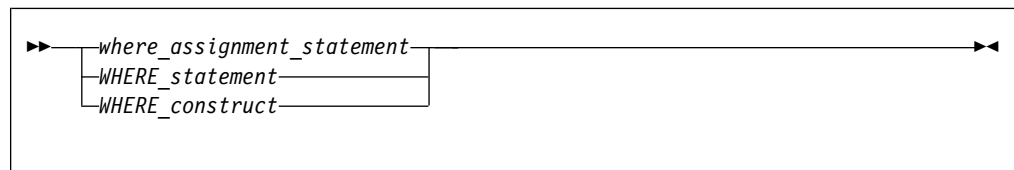
where_assignment_statement
   Is an *assignment_statement*.

masked_ELSEWHERE_block

```
►►─masked_ELSEWHERE_statement──┬─────────────────────┬──────────►◄
                               └─where_body_construct─┘
```
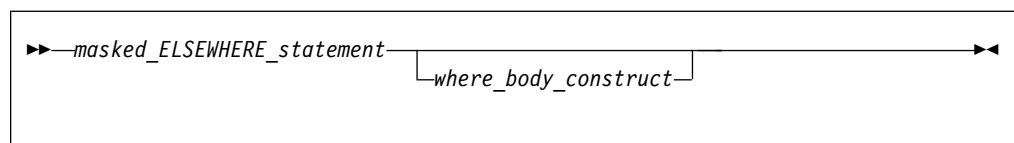
masked_ELSEWHERE_statement
   Is an **ELSEWHERE** statement that specifies a *mask_expr*. See "ELSEWHERE" on page 348 for syntax details.

ELSEWHERE_block

```
►►─ELSEWHERE_statement──┬─────────────────────┬─────────────────►◄
                        └─where_body_construct─┘
```
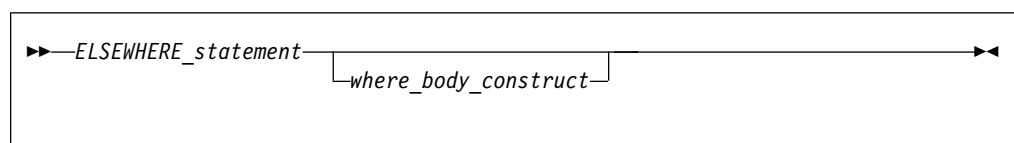
ELSEWHERE_statement

>   Is an **ELSEWHERE** statement that does not specify a *mask_expr*. See "ELSEWHERE" on page 348 for syntax details.

END_WHERE_statement

>   See "END (Construct)" on page 351 for syntax details.

Rules:

- *mask_expr* is a logical array expression.
- In each *where_assignment_statement*, the *mask_expr* and the *variable* being defined must be arrays of the same shape.
- A statement that is part of a *where_body_construct* must not be a branch target statement. Also, **ELSEWHERE**, masked **ELSEWHERE**, and **END WHERE** statements must not be branch target statements.
- A *where_assignment_statement* that is a defined assignment must be an elemental defined assignment.
- The *mask_expr* on the **WHERE** construct statement and all corresponding masked **ELSEWHERE** statements must have the same shape. The *mask_expr* on a nested **WHERE** statement or nested **WHERE** construct statement must have the same shape as the *mask_expr* on the **WHERE** construct statement of the construct in which it is nested.
- If a construct name appears on a **WHERE** construct statement, it must also appear on the corresponding **END WHERE** statement. A construct name is optional on the masked **ELSEWHERE** and **ELSEWHERE** statements in the **WHERE** construct.

Examples

```
REAL, DIMENSION(10) :: A,B,C,D
WHERE (A>0.0)
  A = LOG(A)          ! Only the positive elements of A
                      !   are used in the LOG calculation.
  B = A               ! The mask uses the original array A
                      !   instead of the new array A.
  C = A / SUM(LOG(A)) ! A is evaluated by LOG, but
                      !   the resulting array is an
                      !   argument to a non-elemental
                      !   function. All elements in A will
                      !   be used in evaluating SUM.
END WHERE

WHERE (D>0.0)
  C = CSHIFT(A, 1)    ! CSHIFT applies to all elements in array A,
                      ! and the array element values of D determine
                      ! which CSHIFT expression determines the
                      ! corresponding element values of C.
ELSEWHERE
  C = CSHIFT(A, 2)
END WHERE
END
```

The following example shows an array constructor in a **WHERE** construct statement and in a masked **ELSEWHERE** *mask_expr*:

```
CALL SUB((/ 0, -4, 3, 6, 11, -2, 7, 14 /))

CONTAINS
  SUBROUTINE SUB(ARR)
  INTEGER ARR(:)
  INTEGER N
```

```
      N = SIZE(ARR)

   ! Data in array ARR at this point:
   !
   ! A = | 0 -4 3 6 11 -2 7 14 |

   WHERE (ARR < 0)
      ARR = 0
   ELSEWHERE (ARR < ARR((/(N-I, I=0, N-1)/)))
      ARR = 2
   END WHERE

   ! Data in array ARR at this point:
   !
   ! A = | 2 0 3 2 11 0 7 14 |

   END SUBROUTINE
END
```

The following example shows a nested **WHERE** construct statement and masked **ELSEWHERE** statement with a *where_construct_name*:

```
INTEGER :: A(10, 10), B(10, 10)
...
OUTERWHERE: WHERE (A < 10)
  INNERWHERE: WHERE (A < 0)
     B = 0
  ELSEWHERE (A < 5) INNERWHERE
     B = 5
  ELSEWHERE INNERWHERE
     B = 10
  END WHERE INNERWHERE
ELSEWHERE OUTERWHERE
  B = A
END WHERE OUTERWHERE
...
```

Interpreting masked array assignments

To understand how to interpret masked array assignments, you need to understand the concepts of a *control mask* (m_c) and a *pending control mask* (m_p):

- The m_c is an array of type logical whose value determines which elements of an array in a *where_assignment_statement* will be defined. This value is determined by the execution of one of the following:
  - a **WHERE** statement
  - a **WHERE** construct statement
  - an **ELSEWHERE** statement
  - a masked **ELSEWHERE** statement
  - an **END WHERE** statement

  The value of m_c is cumulative; the compiler determines the value using the mask expressions of surrounding **WHERE** statements and the current mask expression. Subsequent changes to the value of entities in a *mask_expr* have no effect on the value of m_c. The compiler evaluates the *mask_expr* only once for each **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement.

- The m_p is a logical array that provides information to the next masked assignment statement at the same nesting level on the array elements not defined by the current **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement.

The following describes how the compiler interprets statements in a **WHERE**, **WHERE** construct, masked **ELSEWHERE**, **ELSEWHERE**, or **END WHERE** statement. It describes the effect on m_c and m_p and any further behavior of the statements, in order of occurrence.

- **WHERE** statement
  - If the **WHERE** statement is nested in a **WHERE** construct, the following occurs:
    1. m_c becomes m_c **.AND.** *mask_expr*.
    2. After the compiler executes the **WHERE** statement, m_c has the value it had prior to the execution of the **WHERE** statement.
  - Otherwise, m_c becomes the *mask_expr*.
- **WHERE** construct
  - If the **WHERE** construct is nested in another **WHERE** construct, the following occurs:
    1. m_p becomes m_c **.AND.** (**.NOT.** *mask_expr*).
    2. m_c becomes m_c **.AND.** *mask_expr*.
  - Otherwise:
    1. The compiler evaluates the *mask_expr*, and assigns m_c the value of that *mask_expr*.
    2. m_p becomes **.NOT.** *mask_expr*.
- Masked **ELSEWHERE** statement

  The following occurs:
  1. m_c becomes m_p.
  2. m_p becomes m_c **.AND.** (**.NOT.** *mask_expr*).
  3. m_c becomes m_c **.AND.** *mask_expr*.
- **ELSEWHERE** statement

  The following occurs:
  1. m_c becomes m_p. No new m_p value is established.
- **END WHERE** statement

  After the compiler executes an **END WHERE** statement, m_c and m_p have the values they had prior to the execution of the corresponding **WHERE** construct statement.
- *where_assignment_statement*

  The compiler assigns the values of the *expr* that correspond to the true values of m_c to the corresponding elements of the *variable*.

If a non-elemental function reference occurs in the *expr* or *variable* of a *where_assignment_statement* or in a *mask_expr*, the compiler evaluates the function without any masked control; that is, it fully evaluates all of the function's argument expressions and then it fully evaluates the function. If the result is an array and the reference is not within the argument list of a non-elemental function, the compiler selects elements corresponding to true values in m_c for use in evaluating the *expr*, *variable*, or *mask_expr*.

If an elemental intrinsic operation or function reference occurs in the *expr* or *variable* of a *where_assignment_statement* or in a *mask_expr*, and is not within the argument list of a non-elemental function reference, the compiler performs the operation or evaluates the function only for the elements corresponding to true values in m_c.

If an array constructor appears in a *where_assignment_statement* or in a *mask_expr*, the compiler evaluates the array constructor without any masked control and then executes the *where_assignment_statement* or evaluates the *mask_expr*.

The execution of a function reference in the *mask_expr* of a **WHERE** statement is allowed to affect entities in the *where_assignment_statement*. Execution of an **END WHERE** has no effect.

The following example shows how control masks are updated. In this example, *mask1*, *mask2*, *mask3*, and *mask4* are conformable logical arrays, m_c is the control mask, and m_p is the pending control mask. The compiler evaluates each mask expression once.

Sample code (with statement numbers shown in the comments):

```
WHERE (mask1)          ! W1
  WHERE (mask2)        ! W2
    ...                ! W3
  ELSEWHERE (mask3)    ! W4
    ...                ! W5
  END WHERE            ! W6
ELSEWHERE (mask4)      ! W7
...                    ! W8
ELSEWHERE              ! W9
...                    ! W10
END WHERE              ! W11
```

The compiler sets control and pending control masks as it executes each statement, as shown below:

```
Statement W1
    mc = mask1
    mp = .NOT. mask1
Statement W2
    mp = mask1 .AND. (.NOT. mask2)
    mc = mask1 .AND. mask2
Statement W4
    mc = mask1 .AND. (.NOT. mask2)
    mp = mask1 .AND. (.NOT. mask2)
.AND. (.NOT. mask3)
    mc = mask1 .AND. (.NOT. mask2)
.AND. mask3
Statement W6
    mc = mask1
    mp = .NOT. mask1

Statement W7
    mc = .NOT. mask1
    mp = (.NOT. mask1) .AND. (.NOT.
mask4)
    mc = (.NOT. mask1) .AND. mask4
Statement W9
    mc = (.NOT. mask1) .AND. (.NOT.
mask4)
Statement W11
    mc = 0
    mp = 0
```

The compiler uses the values of the control masks set by statements *W2, W4, W7,* and *W9* when it executes the respective *where_assignment_statement*s *W3, W5, W8,* and *W10.*

Migration Tip:

Simplify logical evaluation of arrays

FORTRAN 77 source:
```
INTEGER A(10,10),B(10,10)
  :
  :
DO I=1,10
  DO J=1,10
    IF (A(I,J).LT.B(I,J)) A(I,J)=B(I,J)
  END DO
END DO
END
```

Fortran 90 or Fortran 95 source:
```
INTEGER A(10,10),B(10,10)
  :
  :
WHERE (A.LT.B) A=B
END
```

Examples

```
REAL, DIMENSION(10) :: A,B,C,D
WHERE (A>0.0)
  A = LOG(A)         ! Only the positive elements of A
                     !   are used in the LOG calculation.
  B = A              ! The mask uses the original array A
                     !   instead of the new array A.
  C = A / SUM(LOG(A)) ! A is evaluated by LOG, but
                     !   the resulting array is an
                     !   argument to a non-elemental
                     !   function. All elements in A will
                     !   be used in evaluating SUM.
END WHERE

WHERE (D>0.0)
  C = CSHIFT(A, 1)   ! CSHIFT applies to all elements in array A,
                     ! and the array element values of D determine
                     ! which CSHIFT expression determines the
                     ! corresponding element values of C.
ELSEWHERE
  C = CSHIFT(A, 2)
END WHERE
END
```

The following example shows an array constructor in a **WHERE** construct statement and in a masked **ELSEWHERE** *mask_expr*:

```
CALL SUB((/ 0, -4, 3, 6, 11, -2, 7, 14 /))

CONTAINS
  SUBROUTINE SUB(ARR)
  INTEGER ARR(:)
  INTEGER N

  N = SIZE(ARR)
```

```
   ! Data in array ARR at this point:
   !
   ! A = | 0 -4 3 6 11 -2 7 14 |

   WHERE (ARR < 0)
     ARR = 0
   ELSEWHERE (ARR < ARR((/(N-I, I=0, N-1)/)))
     ARR = 2
   END WHERE

   ! Data in array ARR at this point:
   !
   ! A = | 2 0 3 2 11 0 7 14 |

   END SUBROUTINE
END
```

The following example shows a nested **WHERE** construct statement and masked **ELSEWHERE** statement with a *where_construct_name*:

```
INTEGER :: A(10, 10), B(10, 10)
...
OUTERWHERE: WHERE (A < 10)
  INNERWHERE: WHERE (A < 0)
    B = 0
  ELSEWHERE (A < 5) INNERWHERE
    B = 5
  ELSEWHERE INNERWHERE
    B = 10
  END WHERE INNERWHERE
ELSEWHERE OUTERWHERE
  B = A
END WHERE OUTERWHERE
...
```

FORALL construct

The **FORALL** construct performs assignment to groups of subobjects, especially array elements.

Unlike the **WHERE** construct, **FORALL** performs assignment to array elements, array sections, and substrings. Also, each assignment within a **FORALL** construct need not be conformable with the previous one. The **FORALL** construct can contain nested **FORALL** statements, **FORALL** constructs, **WHERE** statements, and **WHERE** constructs.

-qxlf2003=autorealloc does not apply to **FORALL** constructs. No reallocation of allocatable variables occurs during assignment statements inside a **FORALL** construct.

▶IBM The **INDEPENDENT** directive specifies that the left and right sides of the assignments inside a **FORALL** construct do not overlap. ◀IBM

```
▶▶──FORALL_construct_statement───────────────────────────────────▶◀


▶▶──forall_body─────────────────────────────────────────────────▶◀


▶▶──END_FORALL_statement────────────────────────────────────────▶◀
```

FORALL_construct_statement
        See "FORALL (Construct)" on page 374 for syntax details.

END_FORALL_statement
        See "END (Construct)" on page 351 for syntax details.

forall_body
        is one or more of the following statements or constructs:
            forall_assignment
            WHERE statement (see "WHERE" on page 493)
            WHERE construct (see "WHERE construct" on page 121)
            FORALL statement (see "FORALL" on page 371)
            FORALL construct

forall_assignment
        is either *assignment_statement* or *pointer_assignment_statement*

Any procedures that are referenced in a *forall_body*, including one referenced by a defined operation, defined assignment, or finalization must be pure.

If a **FORALL** statement or construct is nested within a **FORALL** construct, the inner **FORALL** statement or construct cannot redefine any *index_name* used in the outer **FORALL** construct.

Although no atomic object can be assigned to, or have its association status changed in the same statement more than once, different assignment statements within the same **FORALL** construct can redefine or reassociate an atomic object. Also, each **WHERE** statement and assignment statement within a **WHERE** construct must follow these restrictions.

If a *FORALL_construct_name* is specified, it must appear in both the **FORALL** statement and the **END FORALL** statement. Neither the **END FORALL** statement nor any statement within the **FORALL** construct can be a branch target statement.

Interpreting the FORALL construct

1. From the **FORALL** Construct statement, evaluate the *subscript* and *stride* expressions for each *forall_triplet_spec* in any order. All possible pairings of *index_name* values form the set of combinations. For example, given the statement:

   FORALL (I=1:3,J=4:5)

   The set of combinations of I and J is:

       {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}

   The **-1** and **-qnozerosize** compiler options do not affect this step.

2. Evaluate the *scalar_mask_expr* (from the **FORALL** Construct statement) for the set of combinations, in any order, producing a set of active combinations (those that evaluated to .TRUE.). For example, if the mask (I+J.NE.6) is applied to the above set, the set of active combinations is:

    {(1,4),(2,5),(3,4),(3,5)}

3. Execute each *forall_body* statement or construct in order of appearance. For the set of active combinations, each statement or construct is executed completely as follows:

assignment_statement

Evaluate, in any order, all values in the right-hand side *expression* and all subscripts, strides, and substring bounds in the left-hand side *variable* for all active combinations of *index_name* values.

Assign, in any order, the computed *expression* values to the corresponding *variable* entities for all active combinations of *index_name* values. In a *forall_assignment* if *variable* is allocatable, **-qxlf2003=autorealloc** will not cause *variable* to be deallocated and/or allocated.

```
INTEGER, DIMENSION(50) :: A,B,C
INTEGER :: X,I=2,J=49
FORALL (X=I:J)
  A(X)=B(X)+C(X)
  C(X)=B(X)-A(X) ! All these assignments are performed after the
                 ! assignments in the preceding statement
END FORALL
END
```

pointer_assignment_statement

Determine, in any order, what will be the targets of the pointer assignment, and evaluate all subscripts, strides, and substring bounds in the pointer for all active combinations of *index_name* values. If a target is not a pointer, determination of the target does not include evaluation of its value. Pointer assignment never *requires* the value of the righthand side to be determined.

Associate, in any order, all targets with the corresponding pointer entities for all active combinations of *index_name* values.

WHERE statement or construct

Evaluate, in any order, the control mask and pending control mask for each **WHERE** statement, **WHERE** construct statement, **ELSEWHERE** statement, or masked **ELSEWHERE** statement each active combination of *index_name* values, producing a refined set of active combinations for that statement, as described in "Interpreting masked array assignments" on page 123. For each active combination, the compiler executes the assignment(s) of the **WHERE** statement, **WHERE** construct statement, or masked **ELSEWHERE** statement for those values of the control mask that are true for that active combination. The compiler executes each statement in a **WHERE** construct in order, as described previously.

```
INTEGER I(100,10), J(100), X
FORALL (X=1:100, J(X)>0)
  WHERE (I(X,:)<0)
    I(X,:)=0  ! Assigns 0 to an element of I along row X
              ! only if element value is less than 0 and value
              ! of element in corresponding column of J is
  ELSEWHERE   ! greater than 0.
```

```
             I(X,:)=1
        END WHERE
     END FORALL
     END
```

FORALL statement or construct

Evaluate, in any order, the *subscript* and *stride* expressions in the *forall_triplet_spec_list* for the active combinations of the outer **FORALL** statement or construct. The valid combinations are the Cartesian product of combination sets of the inner and outer **FORALL** constructs. The *scalar_mask_expr* determines the active combinations for the inner **FORALL** construct. Statements and constructs for these active combinations are executed.

```
! Same as FORALL (I=1:100,J=1:100,I.NE.J) A(I,J)=A(J,I)

INTEGER A(100,100)
OUTER: FORALL (I=1:100)
  INNER: FORALL (J=1:100,I.NE.J)
    A(I,J)=A(J,I)
  END FORALL INNER
END FORALL OUTER
END
```

Data pointer assignment

Pointer assignment statement causes a pointer to become associated with a target or causes the pointer's association status to become disassociated or undefined.

```
►►──data_pointer_object─────────────────────── => ──target──────►◄
                        │                       (1)  │
                        ├─(bounds_spec_list)─────────┤
                        │                       (2)  │
                        └─(bounds_remapping_list)────┘
```

Notes:

1   Fortran 2003

2   Fortran 2003
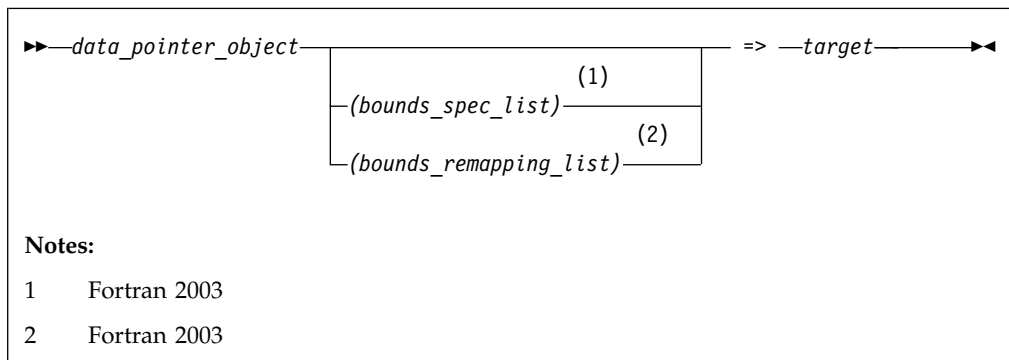
data_pointer_object
        must have the **POINTER** attribute.

target is a variable or expression. If it is a variable, it must have the **TARGET** attribute (or be a subobject of such an object) or the **POINTER** attribute. If it is an expression, it must yield a value that has the **POINTER** attribute.

        If *target* is not unlimited polymorphic, *data_pointer_object* must be type compatible with it and the corresponding kind type parameters must be equal. If *target* is unlimited polymorphic, *data_pointer_object* must be unlimited polymorphic, of sequence derived type, or of a type with the **BIND** attribute.

F2003

bounds_spec

```
►►──lower_bound──:──────────────────────────────────────────◄◄
```

>   *lower_bound*
>   is a scalar integer expression.

bounds_remapping

```
►►──lower_bound : upper_bound────────────────────────────────◄◄
```

>   *lower_bound*
>   is a scalar integer expression.
>
>   *upper_bound*
>   is a scalar integer expression.

F2003

A target must not be an array section with a vector subscript, or a whole assumed-size array.

The size, bounds, and shape of the target of a disassociated array pointer are undefined. No part of such an array can be defined or referenced, although the array can be the argument of an intrinsic inquiry function that is inquiring about association status, argument presence, or a property of the type or type parameters.

IBM A pointer of type byte can only be associated with a target of type byte, **INTEGER(1)**, or **LOGICAL(1)**. **IBM**

F2008 If a *data_pointer_object* is an array with the **CONTIGUOUS** attribute, the *target* must be contiguous. **F2008**

F2003

If *target* is a disassociated pointer, all nondeferred type parameters of the declared type of *data_pointer_object* that correspond to nondeferred type parameters of *target* shall have the same values as the corresponding type parameters of *target*. Otherwise, all nondeferred type parameters of the declared type of *data_pointer_object* must have the same values as the corresponding type parameters of *target*.

If *data_pointer_object* has nondeferred type parameters that correspond to deferred type parameters of *target*, *target* must not be a pointer with undefined association status.

If the *data_pointer_object* is not polymorphic and the *target* is polymorphic with dynamic type that differs from its declared type, the assignment target is the ancestor component of *target* that has the type of *data_pointer_object*. Otherwise, the assignment target is *target*.

If *data_pointer_object* is polymorphic, it assumes the dynamic type of *target*. If *data_pointer_object* is of sequence derived type or a type with the **BIND** attribute, the dynamic type of *target* must be that type.

If you specify *bounds_spec_list*, the number of bounds in the list must be equal to the rank of *data_pointer_object*.

If you specify either a *bounds_spec_list* or a *bounds_remapping_list*, you must not use a **SUBSCRIPTORDER** directive on the pointer

If you specify a *bounds_remapping_list*:
- **F2008** The *target* must be simply contiguous or of rank one. **F2008**
- The *target* must not be a diassociated or undefined pointer, and the size of the target must not be less than the size of the *data_pointer_object*.
- The number of *bounds_remappings* in the list must be equal to the rank of *data_pointer_object*.
- The lower bound of each dimension of the *data_pointer_object* becomes equal to the *lower_bound* you specify in the corresponding *bounds_remapping*.
- The upper bound of each dimension of the *data_pointer_object* becomes equal to the *upper_bound* you specify in the corresponding *bounds_remapping*.
- The extent of each dimension of the *data_pointer_object* is equal to the upper bound of that dimension, minus the lower bound of that dimension, plus 1.
- The elements of the target of *data-pointer-object*, in array element order, are the first **SIZE** (*data-pointer-object*) elements of the target, after any **SUBSCRIPTORDER** directives affect the target.

If you specify a *bounds_spec_list*:
- The number of *bounds_spec* shall equal the rank of *data_pointer_object*.
- If you specify a *bounds_spec_list*, then the *lower_bound* of each dimension of the *data_pointer_object* becomes equal to the *lower_bound* in the corresponding *bounds_spec*.
- The extent of each dimension of the *data_pointer_object* is equal to the extent of the corresponding dimension of the target.
- The *upper bound* of each dimension of the *data_pointer_object* is equal to the *lower bound* of that dimension, plus the extent of that dimension, minus 1.

◀ F2003

If neither *bounds_remapping_list* nor *bounds_spec_list* are specified:

During pointer assignment of an array pointer, the lower bound of each dimension is the result of the **LBOUND** intrinsic function applied to the corresponding dimension of the target. For an array section or array expression that is not a whole array or a structure component, the lower bound is 1. The upper bound of each dimension is the result of the **UBOUND** intrinsic function applied to the corresponding dimension of the target.

Any previous association between a *data_pointer_object* and a target is broken. If *target* is not a pointer, *data_pointer_object* becomes associated with *target*. If *target* is itself an associated pointer, *data_pointer_object* is associated with the target of *target*. If *target* is a pointer with an association status of disassociated or undefined, *data_pointer_object* acquires the same status. If *target* of a pointer assignment is an allocatable object, it must be allocated.

Pointer assignment for a pointer structure component can also occur via execution of a derived-type intrinsic assignment statement or a defined assignment statement.

Related information

- See "ALLOCATE" on page 287 for an alternative form of associating a pointer with a target.
- `F2008` Contiguity `F2008`

Procedure pointer assignment (Fortran 2003)

The procedure pointer assignment statement causes a procedure pointer to become associated with a target or causes the procedure pointer's association status to become disassociated or undefined.

```
►►──proc_pointer_object── => ──proc_target────────────────────────────────────►◄
```

proc_target
>is an expression or a procedure name. If *proc_target* is an expression, it must be a function that returns a procedure pointer. If *proc_target* is a procedure name, it must be the name of an external procedure, a module procedure, a dummy procedure, an intrinsic procedure that can be passed as an actual argument, or another procedure pointer. `F2008` *proc_target* can also be the name of an internal procedure.`F2008` *proc_target* must not be an elemental procedure.

proc_pointer_object
>is a procedure pointer.

If *proc_target* is not a procedure pointer, *proc_pointer_object* becomes associated with *proc_target*. If *proc_target* is a procedure pointer and is associated with a procedure, *proc_pointer_object* becomes associated with the same procedure. If *proc_target* is a pointer with an association status of disassociated or undefined, *proc_pointer_object* acquires the same status.

If the *proc_pointer_object* has an explicit interface, its characteristics must be the same as *proc_target* except that *proc_target* can be pure even if *proc_pointer_object* is not. If the characteristics of *proc_pointer_object* or *proc_target* are such that an explicit interface is required, both *proc_pointer_object* and *proc_target* must have an explicit interface.

If *proc_pointer_object* has an implicit interface and is explicitly typed or referenced as a function, *proc_target* must be a function. If *proc_pointer_object* has an implicit interface and is referenced as a subroutine, *proc_target* must be a subroutine.

If *proc_target* and *proc_pointer_object* are functions, they must have the same type; corresponding type parameters must either be both deferred or have the same value.

If *proc_target* is a specific procedure name that is also a generic name, only the specific procedure is associated with *proc_pointer_object*.

Related information::

• "PROCEDURE declaration (Fortran 2003)" on page 435

Integer pointer assignment (IBM extension)

Integer pointer variables can be:

- Used in integer expressions
- Assigned values as absolute addresses
- Assigned the address of a variable using the **LOC** intrinsic function. (Objects of derived type and structure components must be of sequence-derived type when used with the **LOC** intrinsic function.)

Note: The XL Fortran compiler does not use the size of an object as a multiplier in an arithmetic expression where an integer pointer is an operand.

Examples

```
INTEGER INT_TEMPLATE
POINTER (P,INT_TEMPLATE)
INTEGER MY_ARRAY(10)
DATA MY_ARRAY/1,2,3,4,5,6,7,8,9,10/
INTEGER, PARAMETER :: WORDSIZE=4

P = LOC(MY_ARRAY)
PRINT *, INT_TEMPLATE          ! Prints '1'
P = P + 4;                     ! Add 4 to reach next element
                               !   because arithmetic is byte-based
PRINT *, INT_TEMPLATE          ! Prints '2'

P = LOC(MY_ARRAY)
DO I = 1,10
  PRINT *,INT_TEMPLATE
  P = P + WORDSIZE             ! Parameterized arithmetic is suggested
END DO
END
```

Chapter 7. Execution control

You can control the execution of a program sequence using constructs. Constructs contain statement blocks and other executable statements that can alter the normal execution sequence. This section contains detailed descriptions of the following constructs:

- `F2003` ASSOCIATE `F2003`
- `F2008` BLOCK `F2008`
- DO
- DO WHILE
- IF
- SELECT CASE
- `F2003` SELECT TYPE `F2003`

Detailed syntax diagrams for the constructs in this section can be found by following the links to the associated statements.

For nesting to occur, a construct must be wholly contained within another construct. If a statement specifies a construct name, it applies to that construct. If the statement does not specify a construct name, the statement applies to the innermost construct in which it appears.

In addition to constructs, XL Fortran provides branching as a method for transferring control from one statement to another statement in the same scoping unit.

Statement blocks

A *statement block* consists of a sequence of zero or more executable statements, executable constructs, **FORMAT** statements, or **DATA** statements embedded in another executable construct and are treated as a single unit.

In the same program, you can not transfer control from outside the statement block to within the statement block. You can transfer control within the statement block, or from within the statement block to outside the block. For example, you can have a **GO TO** statement branching to a label that is within a statement block. You cannot branch into a statement block from a **GO TO** statement outside the statement block.

ASSOCIATE Construct (Fortran 2003)

The **ASSOCIATE** construct creates an association between an identifier and a variable, or the value of an expression, during the execution of that construct. The identifier you specify in an **ASSOCIATE** construct becomes an associating entity. You can create multiple associating entities inside a single **ASSOCIATE** construct.

Syntax

```
►►──ASSOCIATE_statement────────────────────────────────────────────►◄


►►──ASSOCIATE_statement_block──────────────────────────────────────►◄


►►──END_ASSOCIATE_statement────────────────────────────────────────►◄
```
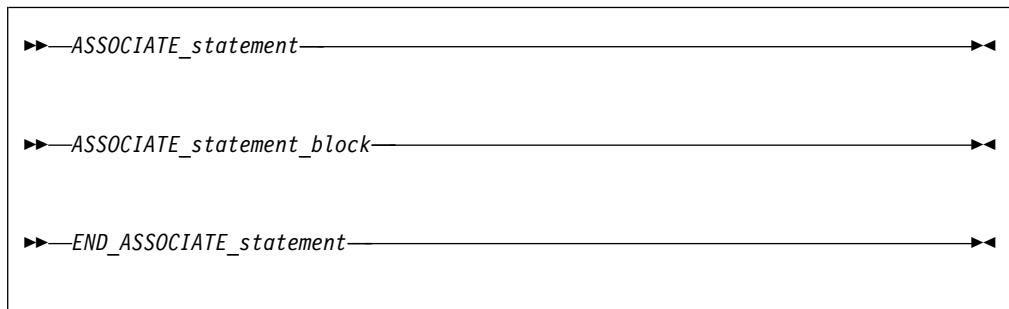
ASSOCIATE_statement
> See "ASSOCIATE (Fortran 2003)" on page 292 for syntax details

END_ASSOCIATE_statement
> See "END (Construct)" on page 351 for syntax details

Execution of an **ASSOCIATE** construct causes execution of an *ASSOCIATE_statement* followed by the *ASSOCIATE_statement_block*. During execution of that block, the construct creates an association with an identifier and the corresponding selector. The associating entity assumes the declared type and type parameters of the selector. The name of the associating entity is an *associate name*. For further information on associate names, see "Associate names" on page 149.

Examples

The following example uses the **ASSOCIATE** construct as a shorthand for a complex expression and renames an existing variable, *MYREAL*. After the end of the **ASSOCIATE** construct, any change within the construct to the value of the associating entity that associates with *MYREAL* is reflected.

```
PROGRAM ASSOCIATE_EXAMPLE

  REAL :: MYREAL, X, Y, THETA, A
  X = 0.42
  Y = 0.35
  MYREAL = 9.1
  THETA = 1.5
  A = 0.4

  ASSOCIATE ( Z => EXP(-(X**2+Y**2)) * COS(THETA), V => MYREAL)
    PRINT *, A+Z, A-Z, V
    V = V * 4.6
  END ASSOCIATE

  PRINT *, MYREAL

END PROGRAM ASSOCIATE_EXAMPLE
```

The expected output is.
```
0.4524610937 0.3475389183 9.100000381

41.86000061
```

BLOCK construct (Fortran 2008)

The **BLOCK** construct defines an executable block that can contain declarations.

Syntax

```
►►—BLOCK_statement—————————————————————————————————————————————————►◄


►►————————————————————————————————————————————————————————————————►◄
    └─specification_part─┘


►►—statement_block—————————————————————————————————————————————————►◄


►►—END_BLOCK_statement—————————————————————————————————————————————►◄
```
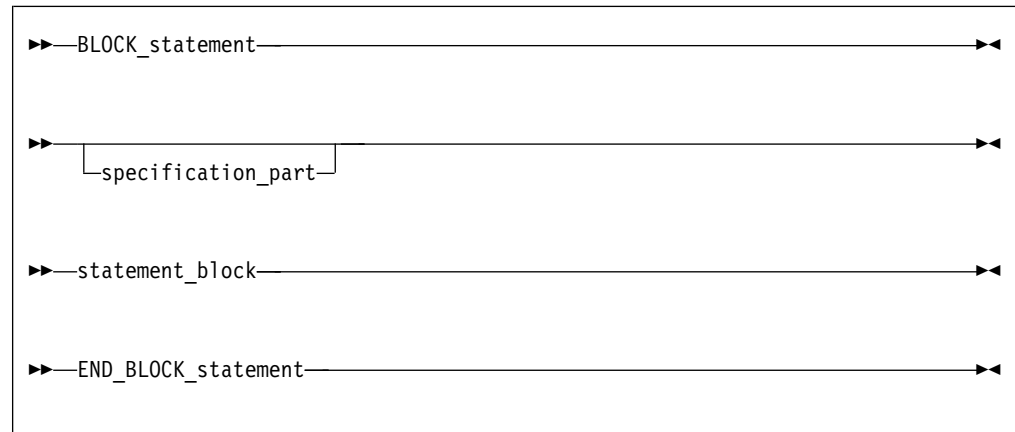
Execution of a **BLOCK** construct causes evaluation of its *specification_part* followed by execution of its *statement_block*.

BLOCK_statement
>   See "BLOCK (Fortran 2008)" on page 299 for syntax details.

END_BLOCK_statement
>   See "END (Construct)" on page 351 for syntax details.

A local variable of a **BLOCK** construct within a pure subprogram cannot have the **SAVE** attribute.

The specification part of the **BLOCK** construct must not contain **COMMON**, **EQUIVALENCE**, **IMPLICIT**, **INTENT**, **NAMELIST**, **OPTIONAL**, statement function, or **VALUE** statements.

The saved entity list of a **SAVE** statement in a **BLOCK** construct must not specify a common block name.

Examples

Example 1: The following example shows that a **BLOCK** construct can be specified with an optional name and nested within another **BLOCK** construct.

```
PROGRAM MyProgram
  INTEGER :: a

  add1 : BLOCK
    INTEGER :: res1
    res1 = a + 1
    ! The BLOCK statement has no BLOCK_construct_name
    BLOCK
      INTEGER :: res2
      res2 = res1 + 1
    END BLOCK
  ! The END BLOCK statement must have the same BLOCK construct name 'add1'
  END BLOCK add1
END PROGRAM MyProgram
```

Example 2: You cannot transfer control from outside a **BLOCK** construct to inside the **BLOCK** construct, except for the return from a procedure call inside the construct. For example:

```
PROGRAM main
  INTEGER :: a

  a = 5
  BLOCK
    INTEGER :: b
    b = a + 2
    ! Program control is returned from a procedure call inside the BLOCK construct
    CALL Sub(b)
  END BLOCK
END PROGRAM main

SUBROUTINE Sub(B)
  INTEGER :: b

  b = b * b
  PRINT *, b
END SUBROUTINE Sub
```

Example 3: The following example shows how an unconditional **GOTO** statement is used to exit a **BLOCK** construct.

```
PROGRAM MyProgram
   INTEGER :: i
   i = 100

   BLOCK
     INTEGER :: i
     i = 1
     ! Before the BLOCK construct is exited, local allocatables are
     ! deallocated, and local finalizable objects are finalized.
     GOTO 10
     i = i + 1
   END BLOCK

10   PRINT *, i
   END PROGRAM MyProgram
```

Related information

- "BLOCK (Fortran 2008)" on page 299
- "SAVE" on page 457
- "Pure procedures" on page 208

DO construct

The **DO** construct specifies the repeated execution of a statement block. Such a repeated block is called a *loop*.

The iteration count of a loop can be determined at the beginning of execution of the **DO** construct, unless it is infinite.

You can curtail a specific iteration with the **CYCLE** statement, and the **EXIT** statement terminates the loop.

```
 ▶▶──DO_statement──────────────────────────────────────────────────▶◀


 ▶▶──statement_block─────────────────────────────────────────────────▶◀


 ▶▶───────┬─END_DO_statement──────┬──────────────────────────────────▶◀
          └─terminal_statement────┘
```
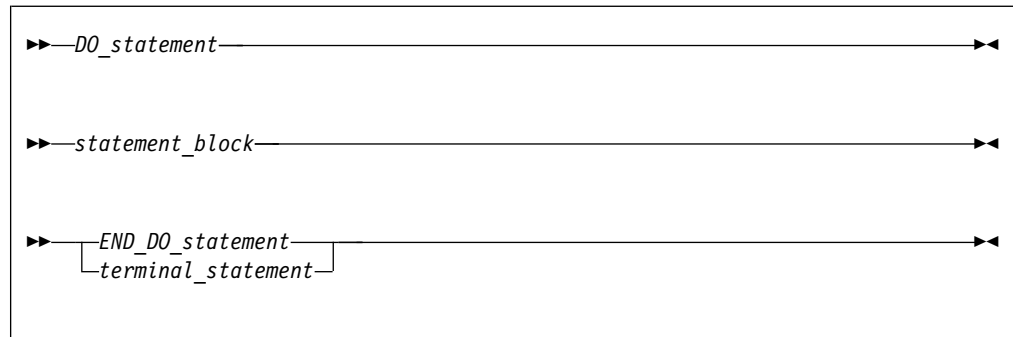
DO_statement
> See "DO" on page 336 for syntax details

END_DO_statement
> See "END (Construct)" on page 351 for syntax details

terminal_statement
> is a statement that terminates the **DO** construct. See below.

If you specify a **DO** construct name on the **DO** statement, you must terminate the construct with an **END DO** statement with the same construct name. Conversely, if you do not specify a **DO** construct name on the **DO** statement, and you terminate the **DO** construct with an **END DO** statement, you must not have a **DO** construct name on the **END DO** statement.

The terminal statement

The terminal statement must follow the **DO** statement and must be executable. See Chapter 11, "Statements and attributes," on page 281 for a listing of statements that can be used as the terminal statement. If the terminal statement of a **DO** construct is a logical **IF** statement, it can contain any executable statement compatible with the restrictions on a logical **IF** statement.

A labeled **DO** statement must be terminated with an **END DO** statement that has a matching statement label. A **DO** statement with no label must be terminated with an unlabeled **END DO** statement.

Nested, labeled **DO**, **DO WHILE**, and ▶F2008▷ **DO CONCURRENT** ◁F2008◁ constructs can share the same terminal statement if the terminal statement is labeled, and if it is not an **END DO** statement.

Range of a DO construct

The range of a **DO** construct consists of all the executable statements following the **DO** statement, up to and including the terminal statement. In addition to the rules governing the range of constructs, you can only transfer control to a shared terminal statement from the innermost sharing **DO** construct.

Active and inactive DO constructs

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

- Its iteration count becomes zero.
- A **RETURN** statement occurs within the range of the **DO** construct.
- Control is transferred to a statement outside the range of the **DO** construct. That statement cannot be an **END DO** statement.

- A subroutine invoked from within the **DO** construct returns, through an alternate return specifier, to a statement that is outside the range of the **DO** construct. That statement cannot be an **END DO** statement.
- An **EXIT** statement that belongs to the **DO** construct executes.
- An **EXIT** statement or a **CYCLE** statement that is within the range of the **DO** construct, but belongs to an outer **DO** or **DO WHILE** construct, executes.
- A **STOP** statement executes or the program stops for any other reason.

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

Executing a DO statement

An infinite **DO** does not have an iteration count limit or a termination condition.

If the loop is not an infinite **DO**, the **DO** statement includes an initial parameter, a terminal parameter, and an optional increment.

1. The initial parameter, m_1, the terminal parameter, m_2, and the increment, m_3, are established by evaluating the **DO** statement expressions (*a_expr1*, *a_expr2*, and *a_expr3*, respectively). Evaluation includes, if necessary, conversion to the type of the **DO** variable according to the rules for arithmetic conversion. (See "Arithmetic conversion" on page 119.) If you do not specify *a_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. The iteration count is determined by the expression:

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

   Note that the iteration count is 0 whenever:

   ```
   m₁ > m₂ and m₃ > 0, or
   m₁ < m₂ and m₃ < 0
   ```

The iteration count cannot be calculated if the **DO** variable is missing. This is referred to as an infinite **DO** construct.

> **IBM** The iteration count cannot exceed $2^{**}31 - 1$ for integer variables of kind 1, 2, or 4, and cannot exceed $2^{**}63 - 1$ for integer variables of kind 8. The count becomes undefined if an overflow or underflow situation arises during the calculation. **IBM**

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

DO execution range

The range of a **DO** construct includes all statements within the statement block. These statements execute until reaching the terminal statement. A **DO** variable

must not become redefined or undefined during execution of the range of a **DO** construct, and only becomes redefined through incremental processing.

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** construct from outside the range. Unless execution of the terminal statement results in a transfer of control, execution continues with incrementation processing.

Incrementation processing

1. The **DO** variable, the iteration count, and the increment of the active **DO** construct whose **DO** statement was most recently executed, are selected for processing.

2. The value of the **DO** variable is increased by the value of m_3.

3. The iteration count is decreased by 1.

4. Execution continues with loop control processing of the same **DO** construct whose iteration count was decremented.

Migration Tip:

- Use **EXIT**, **CYCLE**, and infinite **DO** statements instead of a **GOTO** statement.

FORTRAN 77 source

```
        I = 0
        J = 0
20      CONTINUE
        I = I + 1
        J = J + 1
        PRINT *, I
        IF (I.GT.4) GOTO 10    ! Exiting loop
        IF (J.GT.3) GOTO 20    ! Iterate loop immediately
        I = I + 2
        GOTO 20
10      CONTINUE
        END
```

Fortran 90/95/2003 source:

```
        I = 0 ; J = 0
        DO
          I = I + 1
          J = J + 1
          PRINT *, I
          IF (I.GT.4) EXIT
          IF (J.GT.3) CYCLE
          I = I + 2
        END DO
        END
```

Examples

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO
    READ (5,*) J
    IF (J.LE.I) THEN
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
```

```
      IF (SUM.GT.500) EXIT OUTER
      IF (SUM.GT.100) EXIT INNER
   END DO INNER
   SUM=SUM+I
   I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

DO CONCURRENT construct (Fortran 2008)

With the **DO CONCURRENT** construct, you can specify that individual loop iterations have no interdependencies. The execution order of the iterations can be indeterminate at the beginning of execution of the **DO CONCURRENT** construct.

```
►►──DO_CONCURRENT_statement────────────────────────────────────────►◄


►►──statement_block──────────────────────────────────────────────►◄


►►──┬──END_DO_statement────┬───────────────────────────────────────►◄
    └──terminal_statement──┘
```
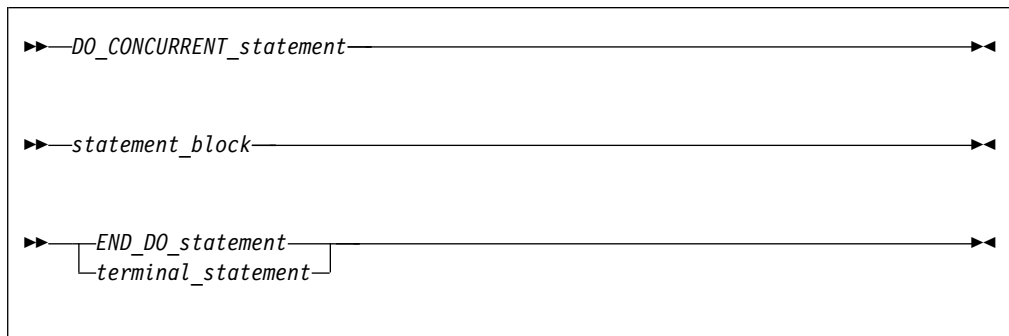
DO_CONCURRENT_statement
> See "DO CONCURRENT (Fortran 2008)" on page 338 for syntax details.

END_DO_statement
> See "END (Construct)" on page 351 for syntax details.

terminal_stmt
> is a statement that terminates the **DO CONCURRENT** construct. See "The terminal statement" on page 139 for details.

Example

The following example shows the usage of the **DO CONCURRENT** construct:

```
INTEGER,PARAMETER::M=100
INTEGER I
Real X(M), Y(M), Z(M)

DO CONCURRENT (I = 1:M)
   Z(I) = SQRT(X(I) + Y(I))
END DO
```

DO WHILE construct

The **DO WHILE** construct specifies the repeated execution of a statement block for as long as the scalar logical expression specified in the **DO WHILE** statement is true. You can curtail a specific iteration with the **CYCLE** statement, and the **EXIT** statement terminates the loop.

```
►►──DO_WHILE_statement───────────────────────────────────────────────────►◄


►►──statement_block──────────────────────────────────────────────────────►◄


►►────┬──END_DO_statement───┬────────────────────────────────────────────►◄
      └──terminal_statement──┘
```
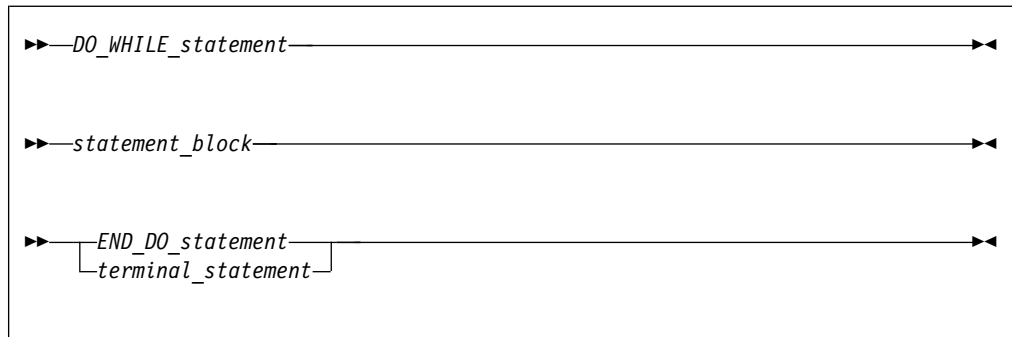
DO_WHILE_statement
> See "DO WHILE" on page 340 for syntax details

END_DO_statement
> See "END (Construct)" on page 351 for syntax details

terminal_stmt
> is a statement that terminates the **DO WHILE** construct. See "The terminal statement" on page 139 for details.

The rules applicable to the **DO** construct names and ranges, active and inactive **DO** constructs, and terminal statements also apply to the **DO WHILE** construct.

Examples
```
I=10
TWO_DIGIT: DO WHILE ((I.GE.10).AND.(I.LE.99))
  J=J+I
  READ (5,*) I
END DO TWO_DIGIT
END
```

IF construct

The **IF** construct selects no more than one of its statement blocks for execution.

```
►►──Block_IF_statement───────────────────────────────────────────────────►◄


►►──statement_block──────────────────────────────────────────────────────►◄


►►──┬──────────────────────┬─────────────────────────────────────────────►◄
    │   ┌◄─────────────┐    │
    └───┴──ELSE_IF_block──┴──┘


►►──┬──────────────┬─────────────────────────────────────────────────────►◄
    └──ELSE_block──┘


►►──END_IF_statement─────────────────────────────────────────────────────►◄
```
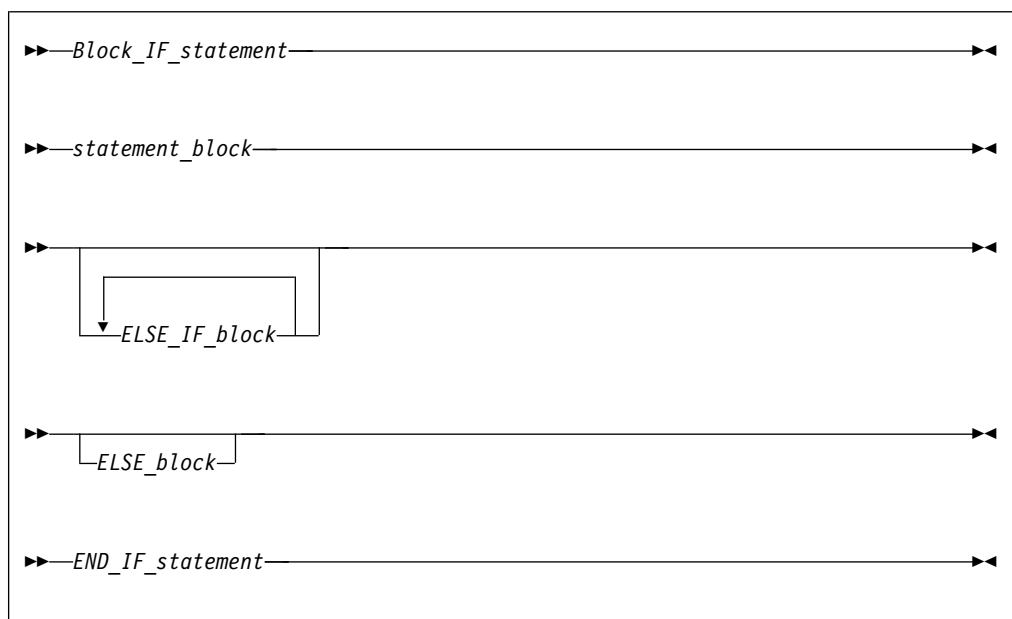
Block_IF_statement
> See "IF (block)" on page 386 for syntax details.

END_IF_statement
> See "END (Construct)" on page 351 for syntax details.

ELSE_IF_block

```
►►──ELSE_IF_statement──────────────────────────────────────────►◄


►►──statement_block────────────────────────────────────────────►◄
```
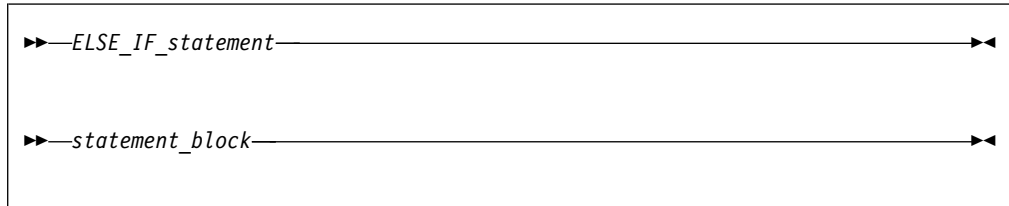
ELSE_IF_statement
> See "ELSE IF" on page 348 for syntax details.

ELSE_block

```
►►──ELSE_statement─────────────────────────────────────────────►◄


►►──statement_block────────────────────────────────────────────►◄
```
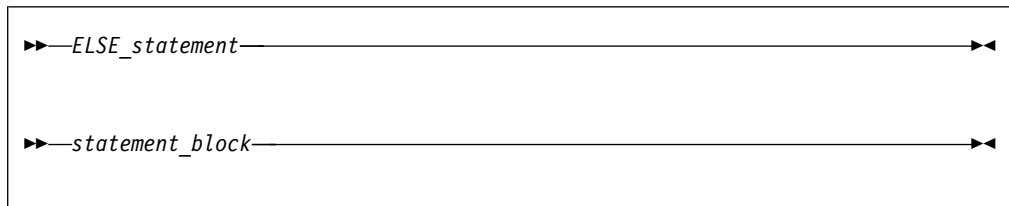
ELSE_statement
> See "ELSE" on page 347 for syntax details.

The scalar logical expressions in an **IF** construct (that is, the block **IF** and **ELSE IF** statements) are evaluated in the order of their appearance until a true value, an **ELSE** statement, or an **END IF** statement is found:

• If a true value or an **ELSE** statement is found, the statement block immediately following executes, and the **IF** construct is complete. The scalar logical expressions in any remaining **ELSE IF** statements or **ELSE** statements of the **IF** construct are not evaluated.

• If an **END IF** statement is found, no statement blocks execute, and the **IF** construct is complete.

If the **IF** construct name is specified, it must appear on the **IF** statement and **END IF** statement, and optionally on any **ELSE IF** or **ELSE** statements. You can branch to an **END IF** statement from only within its **IF** construct.

Examples
```
! Get a record (containing a command) from the terminal

    DO
      WHICHC: IF (CMD .EQ. 'RETRY') THEN      ! named IF construct
          IF (LIMIT .GT. FIVE) THEN           ! nested IF construct
!           Print retry limit exceeded
              CALL STOP
          ELSE
              CALL RETRY
          END IF
      ELSE IF (CMD .EQ. 'STOP') THEN WHICHC   ! ELSE IF blocks
```

```
             CALL STOP
       ELSE IF (CMD .EQ. 'ABORT') THEN
             CALL ABORT
       ELSE WHICHC                            ! ELSE block
!           Print unrecognized command
       END IF WHICHC
     END DO
     END
```

CASE construct

The **CASE** construct has a concise syntax for selecting, at most, one of a number of statement blocks for execution. The case selector of each **CASE** statement is compared to the expression of the **SELECT CASE** statement.
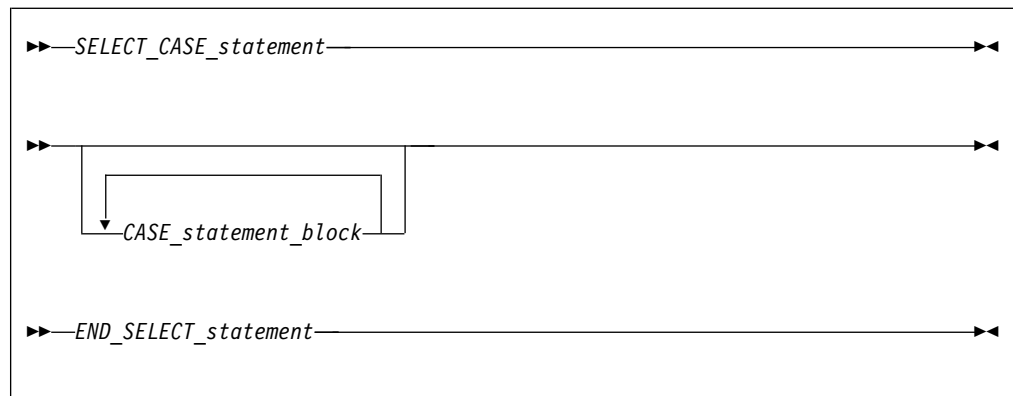
SELECT_CASE_statement
> defines the case expression that is to be evaluated. See "SELECT CASE" on page 459 for syntax details.

END_SELECT_statement
> terminates the **CASE** construct. See "END (Construct)" on page 351 for syntax details.

CASE_statement_block
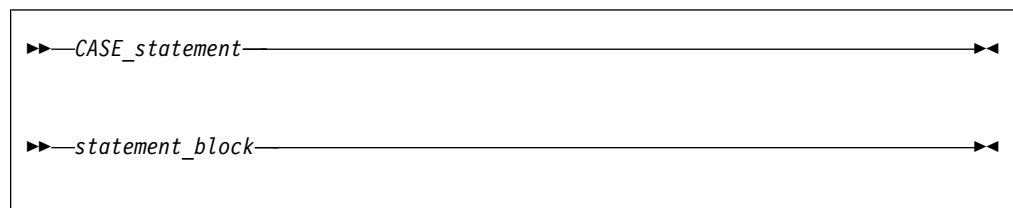
CASE_statement
> defines the case selector, which is a value, set of values, or default case, for which the subsequent statement block is executed. See "CASE" on page 305 for syntax details.

In the construct, each case value must be of the same type as the case expression.

The **CASE** construct executes as follows:
1. The case expression is evaluated. The resulting value is the case index.
2. The case index is compared to the *case_selector* of each **CASE** statement.

- If a match occurs, the statement block associated with that **CASE** statement is executed.
- If no match occurs, but a default *case_selector* exists, the statement block associated with the default *case_selector* is executed.
- Otherwise, no statement block is executed.

3. Execution of the construct is complete and control is transferred to the statement after the **END SELECT** statement.

A **CASE** construct contains zero or more **CASE** statements that can each specify a value range, although the value ranges specified by the **CASE** statements cannot overlap.

A default *case_selector* can be specified by one of the **CASE** statements. A default *CASE_statement_block* can appear anywhere in the **CASE** construct; it can appear at the beginning or end, or among the other blocks.

If a construct name is specified, it must appear on the **SELECT CASE** statement and **END SELECT** statement, and optionally on any **CASE** statements.

You can only branch to the **END SELECT** statement from within the **CASE** construct. A **CASE** statement cannot be a branch target.

Migration Tip:

Use **CASE** in place of block **IF**s.

FORTRAN 77 source
```
      IF (I .EQ.3) THEN
          CALL SUBA()
      ELSE IF (I.EQ. 5) THEN
          CALL SUBB()
      ELSE IF (I .EQ. 6) THEN
          CALL SUBC()
      ELSE
          CALL OTHERSUB()
      ENDIF
      END
```

Fortran 90/95/2003 source:
```
      SELECTCASE(I)
        CASE(3)
          CALL SUBA()
        CASE(5)
          CALL SUBB()
        CASE(6)
          CALL SUBC()
        CASE DEFAULT
          CALL OTHERSUB()
      END SELECT
      END
```

Examples
```
      ZERO: SELECT CASE(N)

      CASE DEFAULT ZERO
          OTHER: SELECT CASE(N) ! start of CASE construct OTHER
            CASE(:-1)
                SIGNUM = -1     ! this statement executed when n≤-1
```

```
                 CASE(1:) OTHER
                      SIGNUM = 1
                 END SELECT OTHER      ! end of CASE construct OTHER
          CASE (0)
            SIGNUM = 0

       END SELECT ZERO
       END
```

Related information
- "CASE" on page 305

SELECT TYPE construct (Fortran 2003)

The **SELECT TYPE** construct selects, at most, one of its constituent blocks for
execution. The selection is based on the dynamic type of an expression. A name is
associated with the expression, in the same way as for the **ASSOCIATE** construct.
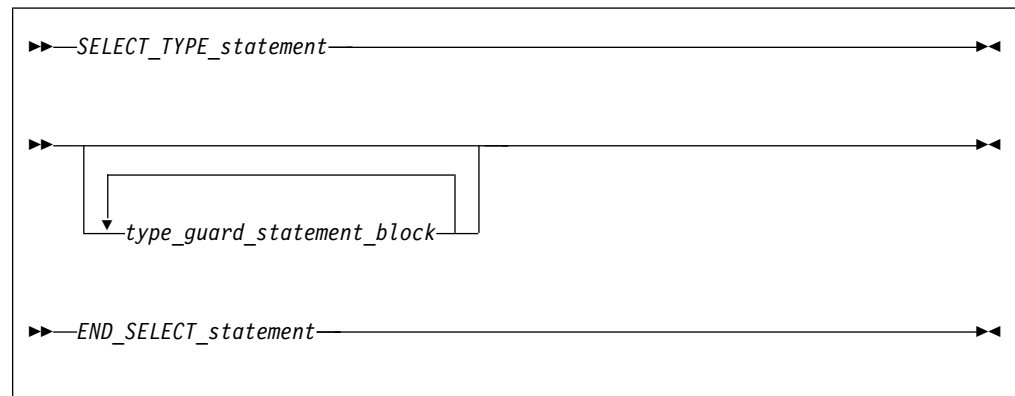
SELECT_TYPE_statement
>   defines the selector expression that is to be evaluated and optionally
>   associates a name (an associate name) with the selector expression. If you
>   do not specify an associate name in the *SELECT_TYPE_statement*, the
>   selector expression must be a named variable. The name of this variable
>   becomes the associate name. Execution of a **SELECT TYPE** construct
>   whose selector is not a variable causes the selector expression to be
>   evaluated. See "SELECT TYPE (Fortran 2003)" on page 460 for syntax
>   details.

type_guard_statement_block
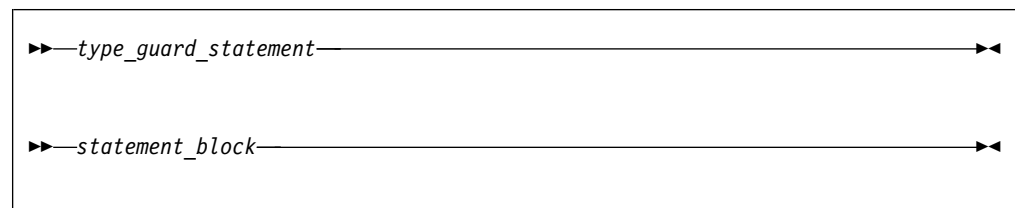
type_guard_statement
>   The dynamic type of the selector expression is compared to the type
>   specified in the *type_guard_statement*. If the rules for type comparison
>   succeed for a particular *type_guard_statement* the subsequent statement
>   block is executed. A type guard statement cannot be a branch target
>   statement. It is permissible to branch to an *end-select-type-stmt* only from

within its **SELECT TYPE** construct. See "Type Guard (Fortran 2003)" on page 483 for syntax details. The other attributes of the associating entity are described in "Associate names" on page 149.

END_SELECT_statement
> terminates the **SELECT TYPE** construct. See "END (Construct)" on page 351 for syntax details.

The block to be executed is selected as follows:

1. If a **TYPE IS** type guard statement matches the selector, the block following that statement is executed. A **TYPE IS** type guard statement matches the selector if the dynamic type and kind type parameter values of the selector are the same as those specified by the statement.

2. Otherwise, if exactly one **CLASS IS** type guard statement matches the selector, the block following that statement is executed.A **CLASS IS** type guard statement matches the selector if the dynamic type of the selector is an extension of the type specified by the statement, and the kind type parameter values specified by the statement are the same as the corresponding type parameter values of the dynamic type of the selector.

3. Otherwise, if several **CLASS IS** type guard statements match the selector, one of these statements must specify a type that is an extension of all the types specified in the others; the block following that statement is executed.

4. Otherwise, if there is a **CLASS DEFAULT** type guard statement, the block following that statement is executed.

If the associating entity follows a **TYPE IS**, **CLASS IS**, or **CLASS DEFAULT** type guard statement, its polymorphism, type, and type parameter are determined as follows:

Table 20. Characteristics of the associating entity in different locations within the construct

Location	Polymorphism	Type	Type parameter
Within the block following a **TYPE IS** type guard statement	Not polymorphic	The type named in the type guard statement	The type parameter values of the selector
Within the block following a **CLASS IS** type guard statement	Polymorphic	The declared type named in the type guard statement	The type parameter values of the selector
Within the block following a **CLASS DEFAULT** type guard statement	Polymorphic	The same declared type as the selector	The type parameter values of the declared type of the selector

Examples

```
PROGRAM p
  TYPE base
    INTEGER::i
  END TYPE

  TYPE,EXTENDS(base)::child
    INTEGER::j
  END TYPE

  CLASS(base), POINTER :: bptr
  TYPE(base), TARGET :: base_target = base(10)
  TYPE(child), TARGET :: child_target = child(20, 30)
```

```
    bptr => base_target

    SELECT type(bptr)
      TYPE IS (base)
      PRINT *, "base type: component value: ", bptr%i
      TYPE IS (child)
      PRINT *, "child type: component values: ", bptr%i, bptr%j
    END SELECT

    bptr => child_target

    SELECT type(bptr)
      TYPE IS (base)
      PRINT *, "base type: component value: ", bptr%i
      TYPE IS (child)
      PRINT *, "child type: component values: ", bptr%i, bptr%j
    END SELECT

END PROGRAM p
```

The output is as follows:

```
base type: component value: 10
child type: component values: 20 30
```

Associate names

Within a **SELECT TYPE** or **ASSOCIATE** construct, an associating entity has the same rank as its associated selector. The lower bound for each dimension is equal to the value of the intrinsic **LBOUND**(*selector*). The upper bound for each dimension is equal to the lower bound plus the extent minus 1.

When the selector is a variable, the associating entity has the **ASYNCHRONOUS** or **VOLATILE** attribute if and only if the selector has the same attribute. If and only if the selector has the **TARGET** or **POINTER** attribute, the associating entity has the **TARGET** attribute. If the associating entity is polymorphic, it assumes the dynamic type and type parameter values of the selector. If the selector has the **OPTIONAL** attribute, it must be present. `F2008` If and only if the selector is contiguous, the associating entity is contiguous. `F2008`

If the selector is not allowed in a variable definition context or is an array with a vector subscript, the associate name must not appear in a variable definition context.

Related information

- `F2008` Contiguity `F2008`

Branching

You can also alter the normal execution sequence by branching. A branch transfers control from one statement to a labeled branch target statement in the same scoping unit. A branch target statement can be any executable statement except a **CASE**, **ELSE**, **ELSE IF**, or type guard statement.

The following statements can be used for branching:
- Assigned **GO TO**

transfers program control to an executable statement, whose statement label is designated in an **ASSIGN** statement. See "GO TO (assigned)" on page 383 for syntax details.

- Computed **GO TO**

  transfers control to possibly one of several executable statements. See "GO TO (computed)" on page 384 for syntax details.

- Unconditional **GO TO**

  transfers control to a specified executable statement. See "GO TO (unconditional)" on page 385 for syntax details.

- Arithmetic **IF**

  transfers control to one of three executable statements, depending on the evaluation of an arithmetic expression. See "IF (arithmetic)" on page 386 for syntax details.

The following input/output specifiers can also be used for branching:

- the **END=** end-of-file specifier

  transfers control to a specified executable statement if an endfile record is encountered (and no error occurs) in a **READ** statement.

- the **ERR=** error specifier

  transfers control to a specified executable statement in the case of an error. You can specify this specifier in the **BACKSPACE**, **ENDFILE**, **REWIND**, **CLOSE**, **OPEN**, **READ**, **WRITE**, and **INQUIRE** statements.

- the **EOR=** end-of-record specifier

  transfers control to a specified executable statement if an end-of-record condition is encountered (and no error occurs) in a **READ** statement.

CONTINUE statement

Execution of a CONTINUE statement has no effect. For more information refer to "CONTINUE" on page 326.

STOP statement

Execution of a **STOP** statement causes normal termination of execution of the program. For more information, see "STOP" on page 464.

ERROR STOP statement (Fortran 2008)

Execution of an **ERROR STOP** statement causes error termination of execution of the program. For more information, see ERROR STOP.

Chapter 8. Program units and procedures

Scope

Scoping unit

A program unit consists of a set of nonoverlapping scoping units. A *scoping unit* is that portion of a program unit that has its own scope boundaries. It is one of the following:

- A derived-type definition.
- **F2008** A **BLOCK** construct (not including any nested **BLOCK** constructs, derived-type definitions, and interface bodies within it). **F2008**
- A procedure interface body (not including any derived-type definitions and interface bodies within it).
- A program unit, module subprogram, or internal subprogram (not including derived-type definitions, **F2008** **BLOCK** constructs **F2008** , interface bodies, module subprograms, and internal subprograms within it).

A host scoping unit is the scoping unit that immediately surrounds another scoping unit. For example, in the following diagram, the host scoping unit of the internal function C is the scoping unit of the main program A. **F2003** Using the **IMPORT** statement, an interface body rather than a module procedure interface body can also access names from its host. **F2003**

Host association is the method by which an internal subprogram, module subprogram, derived-type definition, or **F2008** submodule **F2008** accesses names from its host.

```
PROGRAM A
   INTEGER A1
   CONTAINS

      SUBROUTINE B
         REAL B1
      END SUBROUTINE B


      FUNCTION C ( )
         REAL C1
      END FUNCTION C

END PROGRAM A
```

scope of variable B1

scope of variable C1

scope of variable A1 (not including scope of B1 and C1)
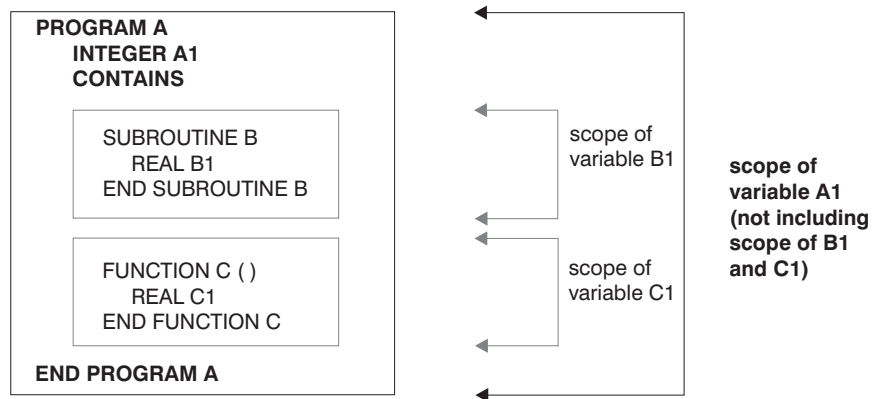
Entities with scope

Entities that have scope are:

- A name (see below)
- A label (local entity)
- An external input/output unit number (global entity)
- An operator symbol. Intrinsic operators are global entities, while defined operators are local entities.
- An assignment symbol (global entity)

If the scope is an executable program, the entity is called a *global entity*. If the scope is a scoping unit, the entity is called a *local entity*. If the scope is a statement or part of a statement, the entity is called a *statement entity*. If the scope is a construct, the entity is called a *construct entity*.

Global entity

Global entities are:
- Common blocks
- **IBM** **CRITICAL** *lock_names* **IBM**
- External input or output units
- External procedures
- **F2003** Entities that have binding labels **F2003**
- Pending data transfer operations
- Program units

You cannot identify an entity of the program by using more than one binding label. If a name identifies a global entity, **F2003** it cannot be the same as any binding label in the same executable program, **F2003** and it cannot be used to identify any other global entity in the same executable program unless that entity is an intrinsic module.

See Conventions for XL Fortran external names in the *XL Fortran Optimization and Programming Guide* for details on restrictions on names of global entities.

Local entity

Entities of the following categories are local entities of the scoping unit where they are defined:

Category 1

> Named variables that are not statement entities, module procedures, named constants, derived types, construct names, generic names, statement functions, internal subprograms, dummy procedures, intrinsic procedures, namelist group names, external procedures with binding labels, abstract interfaces, intrinsic modules, external procedures accessed by a **USE** statement, or statement labels.

Category 2

> Type parameters, components, and type-bound procedure bindings of a derived type definition where each derived type definition has its own category.
>
> - A derived type parameter name, including all that are inherited from its parent type, has the same scope as the derived type being defined.
> - A component name has the same scope as the type of which it is a component. You can specify a name within a component designator of a structure of that type, or as a component keyword in a structure constructor for that type.
> - A binding name of a procedure has the same scope as the type. It can appear only as the binding-name in a procedure reference. A generic binding for which the generic-spec is not a generic name has a scope that consists of all scoping units in which an entity of that type is accessible.
>
> If the derived type is defined in a module and contains the **PRIVATE** statement, the type and its components are accessible in any of the defining module's subprograms by host association. If the accessing

scoping unit accesses this type by use association, that scoping unit, and any scoping unit that accesses the entities of that scoping unit by host association can access the derived types, and only those components with the **PUBLIC** attribute.

Category 3
Argument keywords (in a separate category for each procedure with an explicit interface).

A dummy argument name in an internal procedure, module procedure, or procedure interface block has a scope as an argument keyword of the scoping unit of its host. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or procedure interface block is accessible in another scoping unit by use association or host association, the argument keyword is accessible for procedure references for that procedure in that scoping unit.

Category 4
Common blocks that have binding labels.

Rules

In a scoping unit, a name that identifies a local entity of one category can be used to identify a local entity of another category. Such a name must not be used to identify another local entity of the same category, except in the case of generic names. A generic name of an interface block can be the same as any of the following names:

- A derived type name in the same scope.
- A specific procedure name in the interface block.
- Any accessible generic name.
- Any generic intrinsic procedure name.

In a scoping unit, a name that identifies a global entity cannot be used to identify a local entity of category 1 or category 4 in the same scoping unit, except that the name identifies any of the following items:

- A common block.
- An external function whose name is also a generic name. For details, see "Resolution of procedure references" on page 206.
- An accessible data entity as `use-name` of a `rename` in a **USE** statement.
- An external function and the inclusive scope is its defining subprogram.

IBM A name declared to be a derived type can have the same name as another local entity of category 1 of that scoping unit that is not a derived-type. In this case, the structure constructor for that type is not available in that scope. Similarly, a local entity of category 1 is accessible through host association or use association, even if there is another local entity of category 1 accessible in that scope, if both of the following conditions are satisfied:

- One of the two entities is a derived type and the other is not.
- In the case of host association, the derived type is accessible via host association. For example, given a module M, a program unit P, and an internal subprogram or module subprogram S nested in P, if you have an entity named T1 declared in M that is accessed by use association in P (or in S), you can declare another entity in P (or in S, respectively) with the same name T1, so long as one of the two is a derived type. If you have an entity named T2 accessible in P, and an entity named T2 declared in S, then the T2 accessible in P is accessible in S if the

T2 in P is a derived type. If the T2 in P was not a derived type, it would not be accessible in S if S declared another T2 (of derived type or not).

The structure constructor for that type will not be available in that scope. A local entity of category 1 in a scope that has the same name as a derived type accessible in that scope must be explicitly declared in a declaration statement in that scope.

◀ IBM

If two local entities of category 1, one of which is a derived type, have the same name identifier and are accessible in a scoping unit, any **PUBLIC** or **PRIVATE** statement that specifies the name of the entities applies to both entities. The **PUBLIC** or **PRIVATE** attribute that is specified in the derived type declaration statement also applies to both entities.

If the name of the entities is specified in a **VOLATILE** statement, the entity or entities declared in that scope have the volatile attribute. If the two entities are public entities of a module, any rename on a **USE** statement that references the module and specifies the names of the entities as the *use_name* applies to both entities.

A common block name in a scoping unit can be the name of any local entity other than a named constant or intrinsic procedure. The name is recognized as the common block entity only when the name is delimited by slashes in a **BIND**, **COMMON**, **VOLATILE**, or **SAVE** statement. If it is not, the name identifies the local entity. An intrinsic procedure name can be the name of a common block in a scoping unit that does not reference the intrinsic procedure. In this case, the intrinsic procedure name is not accessible.

The only way that an external function name can also be a local entity is that the external function name is the function result name.

If a scoping unit contains a local entity of category 1 with the same name as an intrinsic procedure, the intrinsic procedure is not accessible in that scoping unit.

Statement entity

The following items are statement entities:

- Name of a statement function dummy argument.
    SCOPE: Scope of the statement in which it appears.
- Name of a variable that appears as the **DO** variable of an implied-**DO** in a **DATA** statement or array constructor.
    SCOPE: Scope of the implied-**DO** list.

Except for a common block name or scalar variable name, the name of a global entity or local entity of category 1 that is accessible in the scoping unit of a statement or construct must not be the name of a statement or construct entity of that statement or construct. Within the scope of a statement or construct entity, another statement or construct entity must not have the same name.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the statement function.

If the name of a global or local entity accessible in the scoping unit of a statement or construct is the same as the name of a statement or construct entity in that statement or construct, the name is interpreted within the scope of the statement or

construct entity as that of the statement or construct entity. Elsewhere in the scoping unit, including parts of the statement or construct outside the scope of the statement or construct entity, the name is interpreted as that of the global or local entity.

If a statement or construct entity has the same name as an accessible name that denotes a variable, constant, or function, the statement or construct entity has the same type and type parameters as the variable, constant or function. Otherwise, the type of the statement or construct entity is determined through the implicit typing rules in effect. If the statement entity is the **DO** variable of an implied-**DO** in a **DATA** statement, the variable cannot have the same name as an accessible named constant.

Related information:

"Local entity" on page 152

"Global entity" on page 152

Statement and construct entity

The following is a statement and/or construct entity:

- Name of a variable that appears as an *index_name* in a **FORALL** statement or **FORALL** construct.
  - SCOPE: Scope of the **FORALL** statement or construct.

The only attributes held by the **FORALL** statement or construct entity are the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the **FORALL**. It is type integer.

Except for a common block name or a scalar variable name, a name that identifies a global entity or a local entity of category 1, accessible in the scoping unit of a **FORALL** statement or construct, must not be the same as the *index_name*. Within the scope of a **FORALL** construct, a nested **FORALL** statement or **FORALL** construct must not have the same *index_name*.

If the name of a global or local entity accessible in the scoping unit of a **FORALL** statement or construct is the same as the *index_name*, the name is interpreted within the scope of the **FORALL** statement or construct as that of the *index_name*. Elsewhere in the scoping unit, the name is interpreted as that of the global or local entity.

Related information:

"Local entity" on page 152

"Global entity" on page 152

Construct entity (Fortran 2003)

See the following construct entity:

- The associate name of an **ASSOCIATE** construct.
  - SCOPE: Scope of the block of the **ASSOCIATE** construct.
- The associate name of a **SELECT TYPE** construct.
  - SCOPE: (Separate) Scope of each block of the **SELECT TYPE** construct.

F2008

- An entity that is explicitly declared in the specification part of a **BLOCK construct**, other than only in **ASYNCHRONOUS** and **VOLATILE** statements.
  - SCOPE: Scope of the **BLOCK** construct.

F2008

If the name of a global or local entity accessible in the scoping unit of an **ASSOCIATE** or **SELECT TYPE** construct is the same as an associate name, the name is interpreted within the blocks of an **ASSOCIATE** or **SELECT TYPE** construct as that of the associate name. Elsewhere in the scoping unit, the name is interpreted as the global and local entities.

Examples

F2008

Example 1: In the following example, the **ASYNCHRONOUS** statement does not define a new variable *a*. It merely gives variable *a*, defined in the outer scope, the **ASYNCHRONOUS** attribute for the duration of the **BLOCK** construct scope.

```
PROGRAM foo
  INTEGER :: a

  BLOCK
    ! This a is the same as the a declared outside the BLOCK construct.
    ! It merely gives variable a, defined in the outer scope, the ASYNCHRONOUS
    ! attribute for the duration of the BLOCK construct scope.
    ASYNCHRONOUS :: a
  END BLOCK
END PROGRAM foo
```

Example 2: In the following example, variable *a* is a construct entity for the **BLOCK** construct, because there is no *a* declared outside the **BLOCK** construct.

```
PROGRAM foo
  BLOCK
    ! This a is a local entity since there is no a in the outer scope.
    INTEGER, ASYNCHRONOUS :: a
  END BLOCK
END PROGRAM foo
```

F2008

Association

Association exists if the same data can be identified with different names in the same scoping unit, or if the same data can be accessed in different scoping units of the same executable program. See "Argument association" on page 195 for information on argument association in procedures and functions.

Host association

Host association allows an internal subprogram, module subprogram, interface body, derived-type definition, or **F2008** submodule **F2008** to access named entities that exist in its host. In interface bodies, entities cannot be accessed by host association unless they are made accessible by an **IMPORT** statement Accessed entities have the same attributes and are known by the same name as they are in the host.

A name that is specified with the **EXTERNAL** attribute is a global name. Any entity in the host scoping unit that has this name as its nongeneric name is inaccessible by that name and by host association.

The following list of entities are local within a scoping unit when declared or initialized in that scoping unit:

• A variable name in a **COMMON** statement or initialized in a **DATA** statement

- An array name in a **DIMENSION** statement
- A name of a derived type
- An object name in a type declaration, **EQUIVALENCE**, **POINTER**, **ALLOCATABLE**, **SAVE**, **TARGET**, **AUTOMATIC**, integer **POINTER**, **STATIC**, or **VOLATILE** statement

  Note: VOLATILE is controlled by compiler option **-qxlf2003**. For more information see the *XL Fortran Compiler Reference*.
- A named constant in a **PARAMETER** statement
- A namelist group name in a **NAMELIST** statement
- A generic interface name or a defined operator
- An intrinsic procedure name in an **INTRINSIC** statement
- A function name in a **FUNCTION** statement, statement function statement, or type declaration statement
- A result name in a **FUNCTION** statement or an **ENTRY** statement
- A subroutine name in a **SUBROUTINE** statement
- An entry name in an **ENTRY** statement
- A dummy argument name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement
- The name of a named construct
- The name of an entity declared by an interface body or **PROCEDURE** declaration statement

Entities in the host scoping unit that have the same name as a local entity are not accessible by host association.

A local entity must not be referenced or defined before the **DATA** statement when:
1. An entity is local to a scoping unit only because it is initialized in a **DATA** statement, and
2. An entity in the host has the same name as this local entity.

If a derived-type name of a host is inaccessible, structures of that type or subobjects of such structures are still accessible.

If a subprogram gains access to a pointer (or integer pointer) by host association, the pointer association that exists at the time the subprogram is invoked remains current within the subprogram. This pointer association can be changed within the subprogram. The pointer association remains current when the procedure finishes executing, except when this causes the pointer to become undefined, in which case the association status of the host-associated pointer becomes undefined. For more information on events that cause definition and undefinition of variables, see "Definition status of variables" on page 21.

The host scoping unit of an internal or module subprogram can contain the same use-associated entities.

F2003

Host associated entities are known by the same name and have the same attributes as in the host, except that an accessed entity may have the **VOLATILE** or **ASYNCHRONOUS** attribute even if the host entity does not. In an internal or module procedure, if a variable that is accessible via host association is specified in

an **ASYNCHRONOUS** or **VOLATILE** statement, that host variable is given the **ASYNCHRONOUS** or **VOLATILE** attribute in the local scope.

Note: VOLATILE is controlled by compiler option **-qxlf2003**.

F2003

F2008 A submodule has access to the entities that are declared and defined in its ancestor module or submodules by host association. The declarations and definitions of entities in a submodule are accessible by host association in its descendant submodules. F2008

Examples

```
SUBROUTINE MYSUB
TYPE DATES                 ! Define DATES
  INTEGER START
  INTEGER END
END TYPE DATES
CONTAINS
  INTEGER FUNCTION MYFUNC(PNAME)
  TYPE PLANTS
    TYPE (DATES) LIFESPAN    ! Host association of DATES
    CHARACTER(10) SPECIES
    INTEGER PHOTOPER
  END TYPE PLANTS
  END FUNCTION MYFUNC
END SUBROUTINE MYSUB
```

Related information

- -qinfo=HOSTASSOCiation

Use association

Use association occurs when a scoping unit accesses the entities of a module with the **USE** statement. You can rename use-associated entities for use in the local scoping unit. The association is in effect for the duration of the executable program. For details, see "USE" on page 483.

F2003

A **USE** associated entity may have the **ASYNCHRONOUS** or **VOLATILE** attribute in the local scoping unit even if the associated module entity does not.

Note: VOLATILE is controlled by compiler option **-qxlf2003**. For more information: *XL Fortran Compiler Reference*.

F2003

Example

```
MODULE m
  CONTAINS
    SUBROUTINE printchar(x)
      CHARACTER(20) x
      PRINT *, x
    END SUBROUTINE
END MODULE

PROGRAM main
```

```
   USE m                   ! Accesses public entities of module m
   CHARACTER(20) :: name = 'George'
   CALL printchar(name)  ! Calls printchar from module m
END PROGRAM
```

Construct Association

Construct association establishes an association between each selector and the corresponding associate name of the construct. Each associate name remains associated with the corresponding selector throughout the execution of the executed block. Within the block, each selector is known by and may be accessed by the corresponding associate name. Construct termination terminates the association as well. See the **ASSOCIATE** and **SELECT TYPE** constructs for more information.

Pointer association

A target that is associated with a pointer can be referenced by a reference to the pointer. This is called pointer association.

A pointer always has an association status:

Associated

- The **ALLOCATE** statement successfully allocates the pointer, which has not been subsequently disassociated or undefined.

  ```
  ALLOCATE (P(3))
  ```

- The pointer is pointer-assigned to a target that is associated or has the **TARGET** attribute and, if allocatable, is allocated.

  ```
  P => T
  ```

Disassociated

- The pointer is nullified by a **NULLIFY** statement or by the **-qinit=f90ptr** option. See **-qinit** in the *XL Fortran Compiler Reference*.

  ```
  NULLIFY (P)
  ```

- **F2003** The pointer is an ultimate component of an object with default initialization specified for the component and:
  - a procedure is invoked with this object as an actual argument corresponding to a nonpointer, nonallocatable dummy argument with **INTENT(OUT)**,
  - a procedure with the object as an unsaved nonpointer, nonallocatable local object that is not accessed by use or host association is invoked,
  - this object is allocated, or
  - **F2008** execution enters a **BLOCK** construct, and the object is an unsaved, nonpointer, nonallocatable, local variable of the BLOCK construct,

    ```
    TYPE DT
       INTEGER, POINTER :: POINT => NULL()
    END TYPE

    BLOCK
      TYPE(DT) DT1  ! DT1%POINT becomes disassociated here
    END BLOCK
    ```

◄ **F2008**

◄ **F2003**

- The pointer is successfully deallocated.

```
DEALLOCATE (P)
```

- The pointer is pointer-assigned to a disassociated pointer.

```
NULLIFY (Q); P => Q
```

Undefined

- Initially (unless the **-qinit=f90ptr** option is specified)
- **F2003** The pointer is an ultimate component of an object, default
  initialization is not specified for the component, and a procedure is
  invoked with this object as an actual argument corresponding to a
  dummy argument with **INTENT(OUT)**, or a procedure is invoked with
  the pointer as an actual argument corresponding to a pointer dummy
  argument with **INTENT(OUT)**. ◄ **F2003**
- If it is pointer-assigned to a pointer whose association status is
  undefined.
- If its target was deallocated other than through the pointer.

```
POINTER P(:), Q(:)
ALLOCATE (P(3))
Q => P
DEALLOCATE (Q)    ! Deallocate target of P through Q.
                  ! P is now undefined.
END
```

- If the execution of a **RETURN** or **END** statement causes the pointer's
  target to become undefined.
- After the execution of a **RETURN** or **END** statement in a procedure
  where the pointer was declared or accessed, except for objects described
  in item 4 under "Events causing undefinition" on page 24.
- **F2008** The target of the pointer becomes undefined when execution
  exits a **BLOCK** construct.

```
INTEGER, POINTER :: POINT
BLOCK
  INTEGER, TARGET :: TARG = 2
  POINT => TARG
END BLOCK  ! point becomes undefined here
```

◄ **F2008**

- **F2008** The pointer is an unsaved, local pointer of a **BLOCK** construct,
  and the execution of the **BLOCK** construct is complete. ◄ **F2008**

Definition status and association status

The definition status of a pointer is that of its target. If a pointer is associated with
a definable target, the definition status of the pointer can be defined or undefined
according to the rules for a variable.

If the association status of a pointer is disassociated or undefined, the pointer must
not be referenced or deallocated. Whatever its association status, a pointer can
always be nullified, allocated or pointer-assigned. When it is allocated, its
definition status is undefined. When it is pointer-assigned, its association and
definition status are determined by its target. So, if a pointer becomes associated
with a target that is defined, the pointer becomes defined.

Integer pointer association (IBM extension)

An integer pointer that is associated with a data object can be used to reference the data object. This is called integer pointer association.

Integer pointer association can only occur in the following situations:

- An integer pointer is assigned the address of a variable:

```
POINTER (P,A)
P=LOC(B)                 ! A and B become associated
```

- Multiple pointees are declared with the same integer pointer:

```
POINTER (P,A), (P,B)   ! A and B are associated
```

- Multiple integer pointers are assigned the address of the same variable or the address of other variables that are storage associated:

```
POINTER (P,A), (Q,B)
P=LOC(C)
Q=LOC(C)                 ! A, B, and C become associated
```

- An integer pointer variable that appears as a dummy argument is assigned the address of another dummy argument or member of a common block:

```
POINTER (P,A)
   .
   .
CALL SUB (P,B)
   .
   .
SUBROUTINE SUB (P,X)
POINTER (P,Y)
P=LOC(X)                 ! Main program variables A
                         !   and B become associated.
```

Program units, procedures, and subprograms

A program unit is a sequence of one or more lines, organized as statements, comments, and directives. A subprogram is either a function or a subroutine, and is either an internal, external, or module subprogram. Specifically, a program unit can be:

- The main program
- A module
- F2008 A submodule F2008
- A block data program unit
- An external function subprogram
- An external subroutine subprogram

An executable program is a collection of program units consisting of one main program and any number of external subprograms, modules, F2008 submodules F2008 , and block data program units.

A subprogram can be invoked by a main program or by another subprogram to perform a particular activity. When a procedure is invoked, the referenced subprogram is executed.

An external or module subprogram can contain multiple **ENTRY** statements. The subprogram defines a procedure for the **SUBROUTINE** or **FUNCTION** statement, as well as one procedure for each **ENTRY** statement.

An external procedure is defined either by an external subprogram or by a program unit in a programming language other than Fortran.

Main programs, external procedures, block data program units, common blocks, entities with binding labels, modules, and F2008 submodules F2008 are global entities. Internal and module procedures are local entities.

Internal procedures

External subprograms, module subprograms, and main programs can have internal subprograms, whether the internal subprograms are functions or subroutines, as long as the internal subprograms follow the **CONTAINS** statement.

An internal procedure is defined by an internal subprogram. Internal subprograms cannot appear in other internal subprograms. A module procedure is defined by a module subprogram or an entry in a module subprogram. Internal procedures and module procedures are the same as external procedures except that:

- The name of an internal procedure or module procedure is not a global entity.
- An internal procedure must not contain an **ENTRY** statement.
- The name of an internal procedure must not be an argument associated with a dummy procedure.
- The internal procedure has access to host entities by host association.
- F2008 The **BIND** attribute with the **NAME=** specifier is not allowed on an internal procedure. F2008

Migration Tip:

Turn your external procedures into internal subprograms or put them into modules. The explicit interface provides type checking.

FORTRAN 77 source
```
      PROGRAM MAIN
        INTEGER A
        A=58
        CALL SUB(A)      ! A must be passed
      END
      SUBROUTINE SUB(A)
        INTEGER A,B,C    ! A must be redeclared
        C=A+B
      END
```

Fortran 90/95/2003 source:
```
PROGRAM MAIN
  INTEGER :: A=58
  CALL SUB
  CONTAINS
  SUBROUTINE SUB
    INTEGER B,C
    C=A+B          ! A is accessible by host association
  END SUBROUTINE
END
```

Interface concepts

The interface of a procedure determines the form of the procedure reference. The interface consists of:

- The characteristics of the procedure
- The name of the procedure
- The name and characteristics of each dummy argument
- The generic identifiers of the procedure, if any

The characteristics of a procedure:
- Distinguishing the procedure as a subroutine or a function
- Distinguishing each dummy argument either as a data object, dummy procedure, or alternate return specifier

  The characteristics of a dummy data object are its declared type, type parameters (if any), shape `TS 29113` (unless it is assumed-rank) `TS 29113`, intent, whether it is optional, allocatable, `F2003` polymorphic, `F2003` a pointer, a target, or has the `F2003` **VALUE** `F2003` or `F2008` CONTIGUOUS `F2008` attribute. Any dependence on other objects for type parameter or array bound determination is a characteristic. If a shape, size, or character length is assumed or deferred, it is a characteristic.

  The characteristics of a dummy procedure are the explicitness of its interface, its procedure characteristics (if the interface is explicit), and whether it is optional.
- If the procedure is a function, it specifies the characteristics of the result value, specifically:
  – Declared type
  – Any type parameters
  – Rank
  – Whether the result value is a pointer
  – Whether the result value is a procedure pointer
  – Whether the result value is allocatable.
  – `F2003` Whether the result value is polymorphic `F2003`
  – `F2008` Whether the result value is contiguous. `F2008`

  For nonpointer, nonallocatable array results, its shape is a characteristic. Any dependence on other objects for type parameters or array bound determination is a characteristic. If the length of a character object is assumed, this is a characteristic. If type parameters of a function result are deferred, which parameters are deferred is a characteristic.
- Determine whether the procedure is **PURE** or **ELEMENTAL**.
- `F2003` Determine if the procedure has the **BIND** attribute. `F2003`

If a procedure is accessible in a scoping unit, it has an interface that is either explicit or implicit in that scoping unit. The rules are:

Entity	Interface
Dummy procedure	Explicit in a scoping unit if an interface block exists or is accessible, or if an explicit interface is specified by a **PROCEDURE** declaration statement. Implicit in all other cases.
External subprogram	Explicit in a scoping unit other than its own if an interface block exists or is accessible, or if an explicit interface is specified by a **PROCEDURE** declaration statement. Implicit in all other cases.
Recursive procedure with a result clause	Explicit in the subprogram's own scoping unit.

Entity	Interface
Module procedure	Always explicit.
Internal procedure	Always explicit.
Generic procedure	Always explicit.
Intrinsic procedure	Always explicit.
Statement function	Always implicit.

Internal subprograms cannot appear in an interface block or in a **PROCEDURE** declaration statement.

A procedure must not have more than one accessible interface in a scoping unit.

The interface of a statement function cannot be specified in an interface block or in a **PROCEDURE** declaration statement.

Explicit interface

A procedure must have an explicit interface in any of the following cases:

1. A reference to the procedure appears
   - with an argument keyword
   - as a defined assignment (for subroutines only)
   - in an expression as a defined operator (for functions only)
   - as a reference by its generic name
   - in a context that requires it to be pure
2. The procedure has
   - a dummy argument that has the F2003 **ALLOCATABLE**, F2003 **OPTIONAL**, **POINTER**, **TARGET** or F2003 **VALUE** F2003 attributes
   - F2003 a dummy argument that is polymorphic F2003
   - an array-valued result (for functions only)
   - a result whose length type parameter is neither assumed nor constant (for character functions only)
   - a pointer or allocatable result (for functions only)
   - a dummy argument that is an assumed-shape array
3. The procedure is elemental.
4. F2003 The procedure has the **BIND** attribute. F2003

Implicit interface

A procedure has an implicit interface if its interface is not fully known; that is, it has no explicit interface.

Interface blocks

The interface block allows you to specify an explicit interface for external and dummy procedures. You can also use an interface block to define generic identifiers. An interface body in an interface block contains the explicit specific interface for an existing external procedure or dummy procedure. You can also specify the interface for a procedure using a procedure statement.

```
 ►►──INTERFACE_statement─────────────────────────────────────────────►◄



 ►►──┬──────────────────────────────────┬──────────────────────────────►◄
     │ ◄──────────────────────────────┐ │
     └─┬──FUNCTION_interface_body────┬─┘
       ├──SUBROUTINE_interface_body──┤
       └──PROCEDURE_statement────────┘


 ►►──END_INTERFACE_statement─────────────────────────────────────────►◄
```
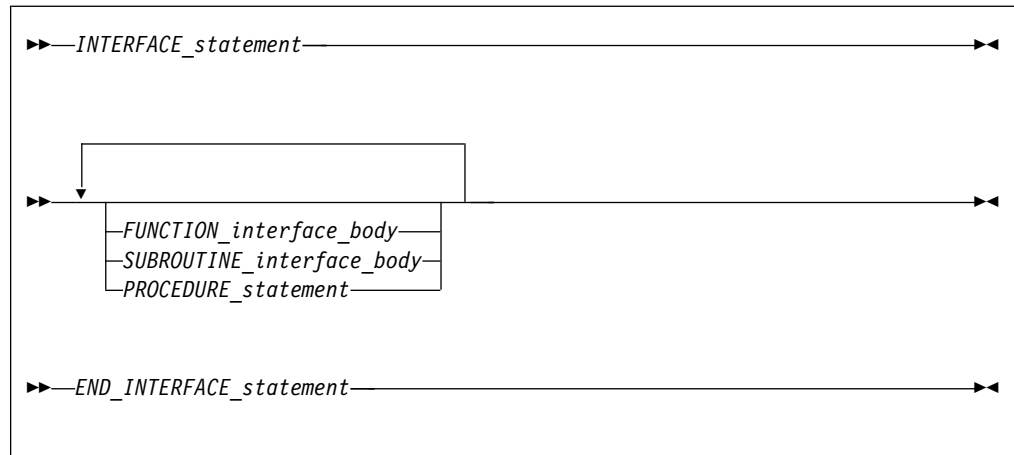
INTERFACE_statement
> See "INTERFACE" on page 405 for syntax details

END_INTERFACE_statement
> See "END INTERFACE" on page 354 for syntax details

PROCEDURE_statement
> See "PROCEDURE" on page 433 for syntax details
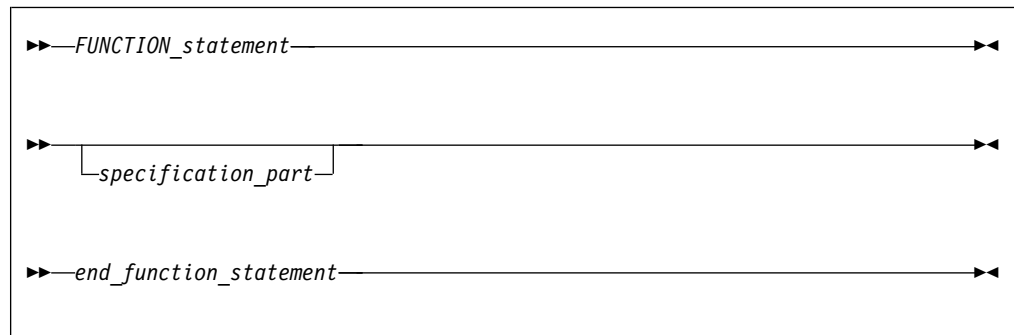
FUNCTION_interface_body

```
 ►►──FUNCTION_statement───────────────────────────────────────────────►◄


 ►►──┬─────────────────────┬─────────────────────────────────────────►◄
     └──specification_part──┘


 ►►──end_function_statement──────────────────────────────────────────►◄
```

SUBROUTINE_interface_body

```
 ►►──SUBROUTINE_statement─────────────────────────────────────────────►◄


 ►►──┬─────────────────────┬─────────────────────────────────────────►◄
     └──specification_part──┘


 ►►──end_subroutine_statement────────────────────────────────────────►◄
```

FUNCTION_statement, SUBROUTINE_statement
> For syntax details, see "FUNCTION" on page 379 and "SUBROUTINE" on
> page 468.

specification_part
> is a sequence of statements from the statement groups numbered **2** , **3** and **5** in "Order of statements and execution sequence" on page 16.

end_function_statement, *end_subroutine_statement*
> For syntax details of both statements, see "END" on page 350.

In an interface body or with a procedure declaration statement, you specify all the characteristics of the procedure or abstract interface. See "Interface concepts" on page 162. The characteristics must be consistent with those specified in the subprogram definition, except that:

1. dummy argument names may be different.
2. you do not have to indicate that a procedure is pure, even if the subprogram that defines it is pure.
3. you can associate a pure actual argument with a dummy procedure that is not pure.
4. when you associate an intrinsic elemental procedure with a dummy procedure, the dummy procedure does not have to be elemental.

The *specification_part* of an interface body can contain statements that specify attributes or define values for data objects that do not determine characteristics of the procedure. Such specification statements have no effect on the interface. Interface blocks do not specify the characteristics of module procedures, whose characteristics are defined in the module subprogram definitions.

An interface body cannot contain **ENTRY** statements, **DATA** statements, **FORMAT** statements, statement function statements, or executable statements. You can specify an entry interface by using the entry name as the procedure name in an interface body.

An interface body does not access named entities by host association unless you specify the F2003 **IMPORT** F2003 statement. It is treated as if it had a host with the default implicit rules. See "Determining Type" on page 18 for a discussion of the implicit rules.

An interface block can be abstract, generic or specific. A generic interface block must specify a generic specification in the **INTERFACE** statement, while an abstract or specific interface block must not specify such a generic specification. See "INTERFACE" on page 405 for details.

The interface bodies within an abstract or specific interface block can contain interfaces for both subroutines and functions.

A generic name specifies a single name to reference all of the procedures in the interface block. At most, one specific procedure is invoked each time there is a procedure reference with a generic name.

The **PROCEDURE** statement is allowed only if the interface block has a generic specification and is contained in a scoping unit where each procedure name is accessible.

A procedure name used in a **PROCEDURE** statement must not have been previously specified in any **MODULE PROCEDURE** statement in any accessible interface block with the same generic identifier.

IBM For an interface to a non-Fortran subprogram, the dummy argument list in the **FUNCTION** or **SUBROUTINE** statement can explicitly specify the passing method. See "Dummy arguments" on page 194 for details. IBM

Examples

```
MODULE M
 CONTAINS
 SUBROUTINE S1(IARG)
   IARG = 1
 END SUBROUTINE S1
 SUBROUTINE S2(RARG)
   RARG = 1.1
 END SUBROUTINE S2
 SUBROUTINE S3(LARG)
   LOGICAL LARG
   LARG = .TRUE.
 END SUBROUTINE S3
END

USE M
INTERFACE SS
  SUBROUTINE SS1(IARG,JARG)
  END SUBROUTINE
  MODULE PROCEDURE S1,S2,S3
END INTERFACE
CALL SS(II)              ! Calls subroutine S1 from M
CALL SS(I,J)             ! Calls subroutine SS1
END

SUBROUTINE SS1(IARG,JARG)
  IARG = 2
  JARG = 3
END SUBROUTINE
```

You can always reference a procedure through its specific interface. If a generic interface exists for a procedure, the procedure can also be referenced through the generic interface.

Within an interface body, if a dummy argument is intended to be a dummy procedure, it must have the **EXTERNAL** attribute or there must be an interface for the dummy argument.

Generic interface blocks

In an **INTERFACE** statement, a generic interface block must specify one of the following:

- a generic name
- defined operator
- defined assignment
- F2003 a derived-type input/output generic specification F2003

The generic name is a single name with which to reference all of the procedures specified in the interface block. It can be the same as any accessible generic name, or any of the procedure names in the interface block.

If two or more generic interfaces that are accessible in a scoping unit have the same local name, they are interpreted as a single generic interface.

Unambiguous generic procedure references

When making a reference to a generic procedure, only one specific procedure is invoked. This section includes rules that ensure a generic reference is unambiguous.

If two procedures in the same scoping unit both define assignment or both have the same defined operator and the same number of arguments, you must specify a dummy argument that corresponds by position in the argument list to a dummy argument of the other that is distinguishable from it. Two dummy arguments are distinguishable if they meet one of the following requirements:

- **F2008** One is a procedure and the other is a data object. **F2008**
- Both are data objects and neither is TKR-compatible with the other.
- **F2008** Both are functions and neither is TKR-compatible with the other. **F2008**
- **F2008** Both are data objects where one has the **ALLOCATABLE** attribute and the other has the **POINTER** attribute but does not have the **INTENT(IN)** attribute. **F2008**
- **F2008** One is a function that returns an array and the other is not known to be a function. **F2008**

Within a scoping unit, if two procedures have the same *dtio_generic_spec*, their *dtv* arguments must be type-incompatible or have different kind type parameters. (For information about *dtio_generic_spec* specifications and the *dtv* argument, see "User-defined derived-type Input/Output procedure interfaces (Fortran 2003)" on page 222).

Within a scoping unit, two procedures that have the same generic name must both be subroutines or both be functions. They must also adhere to the following conditions:

1. One of the procedures contains a non-passed-object dummy argument such that the number of dummy arguments in one procedure that are nonoptional, not passed-object, and with which the dummy argument is TKR-compatible, possibly including the dummy argument itself, exceeds the number of non-passed-object dummy arguments, both optional and nonoptional, in the other procedure that is not distinguishable from the dummy argument.
2. Both procedures have passed-object dummy arguments, which are distinguishable.
3. At least one of the procedures has both:
   a. A nonoptional non-passed-object dummy argument at an effective position such that either the other procedure has no dummy argument at that effective position or the dummy argument at that position is distinguishable from it
   b. A nonoptional non-passed-object dummy argument whose name is such that either the other procedure has no dummy argument with that name or the dummy argument with that name is distinguishable from it.

The dummy argument that disambiguates by position must either be the same as, or occur earlier in the argument list than, the one that disambiguates by name.

The effective position of a dummy argument is its position in the argument list after any passed-object dummy argument has been removed.

When an interface block extends an intrinsic procedure, the rules in this section apply as if the intrinsic procedure consists of a collection of specific procedures, one procedure for each allowed set of arguments. **IBM**

Notes:

- Dummy arguments of type **BYTE** are considered to have the same type as corresponding 1-byte dummy arguments of type **INTEGER(1)**, **LOGICAL(1)**, and character.
- When the **-qintlog** compiler option is specified, dummy arguments of type integer and logical are considered to have the same type as corresponding dummy arguments of type integer and logical with the same kind type parameter.

◀**IBM**

Examples

```
PROGRAM MAIN
INTERFACE A
  FUNCTION AI(X)
    INTEGER AI, X
  END FUNCTION AI
END INTERFACE
INTERFACE A
  FUNCTION AR(X)
    REAL AR, X
  END FUNCTION AR
END INTERFACE
INTERFACE FUNC
  INTEGER FUNCTION FUNC1(I, EXT)
    INTEGER I
    EXTERNAL EXT              ! Here, EXT is a procedure
  END FUNCTION FUNC1
  INTEGER FUNCTION FUNC2(EXT, I)
    INTEGER I
    REAL EXT                  ! Here, EXT is a variable
  END FUNCTION FUNC2
END INTERFACE
EXTERNAL MYFUNC
IRESULT=A(INTVAL)          ! Call to function AI
RRESULT=A(REALVAL)         ! Call to function AR
RESULT=FUNC(1,MYFUNC)      ! Call to function FUNC1
END PROGRAM MAIN

FUNCTION AI(X)
  INTEGER AI, X
  PRINT *," Inside AI()"
  AI = X + 10
END FUNCTION AI

FUNCTION AR(X)
  REAL AR, X
  PRINT *," Inside AR()"
  AR = X - 2.0
END FUNCTION AR

INTEGER FUNCTION FUNC1(I, EXT)
  INTEGER I
  EXTERNAL EXT
  PRINT *," Inside FUNC1()"
  FUNC1 = EXT(I)
END FUNCTION FUNC1

INTEGER FUNCTION MYFUNC (I)
```

```
      INTEGER I
      PRINT *,"    Inside MYFUNC()"
      MYFUNC = I - 100
    END FUNCTION MYFUNC
```

Extending intrinsic procedures with generic interface blocks

A generic intrinsic procedure can be extended or redefined. An extended intrinsic procedure supplements the existing specific intrinsic procedures. A redefined intrinsic procedure replaces an existing specific intrinsic procedure.

When a generic name is the same as a generic intrinsic procedure name and the name has the **INTRINSIC** attribute (or appears in an intrinsic context), the generic interface extends the generic intrinsic procedure.

When a generic name is the same as a generic intrinsic procedure name and the name does not have the **INTRINSIC** attribute (nor appears in an intrinsic context), the generic interface can redefine the generic intrinsic procedure.

A generic interface name cannot be the same as a specific intrinsic procedure name if the name has the **INTRINSIC** attribute (or appears in an intrinsic context).

Examples

```
    PROGRAM MAIN
    INTRINSIC MAX
    INTERFACE MAX                ! Extension to intrinsic MAX
      FUNCTION MAXCHAR(STRING)
        CHARACTER(50) STRING
      END FUNCTION MAXCHAR
    END INTERFACE
    INTERFACE ABS                ! Redefines generic ABS as
      FUNCTION MYABS(ARG)        !   ABS does not appear in
        REAL(8) MYABS, ARG       !   an INTRINSIC statement
      END FUNCTION MYABS
    END INTERFACE
    REAL(8) DARG, DANS
    REAL(4) RANS
    INTEGER IANS,IARG
    CHARACTER(50) NAME
    DANS = ABS(DARG)             ! Calls external MYABS
    IANS = ABS(IARG)             ! Calls intrinsic IABS
    DANS = DABS(DARG)            ! Calls intrinsic DABS
    IANS = MAX(NAME)             ! Calls external MAXCHAR
    RANS = MAX(1.0,2.0)          ! Calls intrinsic AMAX1
    END PROGRAM MAIN
```

Defined operators

A defined operator is a user-defined unary or binary operator, or an extended intrinsic operator (see "Extended intrinsic and defined operations" on page 113). It must be defined by both a function and a generic interface block.

1. To define the unary operation *op* x_1:
   a. A function or entry must exist that specifies exactly one dummy argument, d_1.
   b. **F2003** Either:
      1) the *generic_spec* in an **INTERFACE** statement specifies **OPERATOR** *(op)*, or
      2) there is a generic binding in the declared type of *x1* with a *generic_spec* of **OPERATOR(op)** and there is a corresponding binding to the function in the dynamic type of *x1*. **F2003**

    c. The dynamic type of x_1 is compatible with the type of the dummy argument d_1.

    d. The type parameters, if any, of x_1 must match those of d_1.

    e. Either
- The function is **ELEMENTAL**, or
- The rank of x_1, and its shape, if it is an array, match those of d_1

2. To define the binary operation x_1 *op* x_2:

    a. The function is specified with a **FUNCTION** or **ENTRY** statement that specifies two dummy arguments, d_1 and d_2.

    b. `F2003` Either:
   1) the *generic_spec* in an **INTERFACE** block specifies **OPERATOR** *(op)*, or
   2) there is a generic binding in the declared type of *x1* or *x2* with a *generic_spec* of **OPERATOR(op)** and there is a corresponding binding to the function in the dynamic type of *x1* or *x2*, respectively. `F2003`

    c. The dynamic types of x_1 and x_2 are compatible with the types of the dummy arguments d_1 and d_2, respectively.

    d. The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively.

    e. Either:
- The function is **ELEMENTAL** and x_1 and x_2 are conformable or,
- The ranks of x_1 and x_2 and their shapes, if either or both are arrays, match those of d_1 and d_2, respectively.

3. If *op* is an intrinsic operator, the types or ranks of either x_1 or x_2 are not those required for an intrinsic operation.

4. The *generic_spec* must not specify **OPERATOR** for functions with no arguments or for functions with more than two arguments.

5. Each argument must be nonoptional.

6. The arguments must be specified with **INTENT(IN)**.

7. Each function specified in the interface cannot have a result of assumed character length.

8. If the operator specified is an intrinsic operator, the number of function arguments must be consistent with the intrinsic uses of that operator.

9. A given defined operator can, as with generic names, apply to more than one function, in which case it is generic just like generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent.

10. `IBM` The following rules apply only to extended intrinsic operations:

    a. The type of one of the arguments can only be of type **BYTE** when the type of the other argument is of derived type.

    b. When the **-qintlog** compiler option has been specified for non-character operations, and d_1 is numeric or logical, then d_2 must not be numeric or logical.

    c. When the **-qctyplss** compiler option has been specified for non-character operations, if x_1 is numeric or logical and x_2 is a character constant, the intrinsic operation is performed.

    `IBM`

Examples

```
INTERFACE OPERATOR (.DETERMINANT.)
  FUNCTION IDETERMINANT (ARRAY)
    INTEGER, INTENT(IN), DIMENSION (:,:) :: ARRAY
```

```
        INTEGER IDETERMINANT
      END FUNCTION
    END INTERFACE
    END
```

Defined assignment

A defined assignment is treated as a reference to a subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument.

1. To define the defined assignment $x_1 = x_2$:

   a. The subroutine is specified with a **SUBROUTINE** or **ENTRY** statement that specifies two dummy arguments, d_1 and d_2.

   b. `F2003` Either:

      1) the *generic_spec* of an interface block specifies **ASSIGNMENT (=)**, or

      2) there is a generic binding in the declared type of *x1* or *x2* with a *generic_spec* of **ASSIGNMENT(=)** and there is a corresponding binding to the subroutine in the dynamic type of *x1* or *x2*, respectively. `F2003`

   c. The dynamic types of x_1 and x_2 are compatible with the types of dummy arguments d_1 and d_2, respectively.

   d. The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively.

   e. Either:

      • The subroutine is **ELEMENTAL** and either x_1 and x_2 have the same shape, x_2 is scalar, or

      • The ranks of x_1 and x_2, and their shapes, if either or both are arrays, match those of d_1 and d_2, respectively.

2. **ASSIGNMENT** must only be used for subroutines with exactly two arguments.

3. Each argument must be nonoptional.

4. The first argument must have **INTENT(OUT)** or **INTENT(INOUT)**, and the second argument must have **INTENT(IN)**.

5. The types of the arguments must not be both numeric, both logical, or both character with the same kind parameter.

   `IBM` The type of one of the arguments can only be of type **BYTE** when the type of the other argument is of derived type.

   When the **-qintlog** compiler option has been specified, and d_1 is numeric or logical, then d_2 must not be numeric or logical.

   When the **-qctyplss** compiler option has been specified, if x_1 is numeric or logical and x_2 is a character constant, intrinsic assignment is performed. `IBM`

6. The **ASSIGNMENT** generic specification specifies that the assignment operation is extended or redefined if both sides of the equal sign are of the same derived type.

Examples

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN), DIMENSION(:) :: B
  END SUBROUTINE
END INTERFACE
```

User-defined derived-type Input/Output procedures (Fortran 2003)

User-defined derived-type input/output procedures allow a program to override the default handling of derived-type objects and values in data transfer input/output statements.

There are four interfaces, one for each of the following I/O operations:
- formatted input
- formatted output
- unformatted input
- unformatted output

The four interfaces use a *dtv* type specification, or *dtv_type_spec*. The syntax for the *dtv_type_spec* is as follows:

```
>>----+--TYPE---+---(--derived_type_spec--)---------------------------><
      +--CLASS--+
```
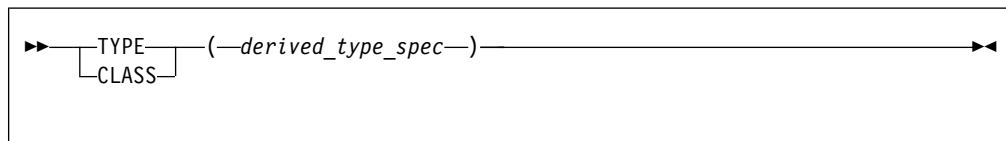
If *derived_type_spec* specifies an extensible type, use the **CLASS** keyword; otherwise, use the **TYPE** keyword.

All length type parameters of *derived_type_spec* must be assumed.

The following table shows the required characteristics of the user-defined procedures for each of the four *dtio_generic_spec* specifications of the interface block or generic binding:

Table 21. Interfaces for user-defined derived-type input/output procedures

dtio_generic_spec	Interface

READ (FORMATTED)	```SUBROUTINE my_read_routine_formatted &```
	```(dtv, unit, iotype, v_list, iostat, iomsg)```
	``` INTEGER, INTENT(IN) :: unit ! unit number```
	``` ! the derived-type value/variable```
	``` dtv_type_spec, INTENT(INOUT) :: dtv```
	``` ! the edit descriptor string```
	``` CHARACTER (LEN=*), INTENT(IN) :: iotype```
	``` INTEGER, INTENT(IN) :: v_list(:)```
	``` INTEGER, INTENT(OUT) :: iostat```
	``` CHARACTER (LEN=*), INTENT(INOUT) :: iomsg```
	```END SUBROUTINE```

READ (UNFORMATTED)	```SUBROUTINE my_read_routine_unformatted &```
	```(dtv, unit, iostat, iomsg)```
	``` INTEGER, INTENT(IN) :: unit```
	``` ! the derived-type value/variable```
	``` dtv_type_spec, INTENT(INOUT) :: dtv```
	``` INTEGER, INTENT(OUT) :: iostat```
	``` CHARACTER (LEN=*), INTENT(INOUT) :: iomsg```
	```END SUBROUTINE```

Table 21. Interfaces for user-defined derived-type input/output procedures (continued)

dtio_generic_spec	Interface

WRITE (FORMATTED)	```
	SUBROUTINE my_write_routine_formatted &
	(dtv, unit, iotype, v_list, iostat, iomsg)
	  INTEGER, INTENT(IN) :: unit
	  ! the derived-type value/variable
	  dtv_type_spec, INTENT(IN) :: dtv
	  ! the edit descriptor string
	  CHARACTER (LEN=*), INTENT(IN) :: iotype
	  INTEGER, INTENT(IN) :: v_list(:)
	  INTEGER, INTENT(OUT) :: iostat
	  CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
	 END SUBROUTINE
	``

WRITE (UNFORMATTED)	```
	SUBROUTINE my_write_routine_unformatted &
	(dtv, unit, iostat, iomsg)
	  INTEGER, INTENT(IN) :: unit
	  ! the derived-type value/variable
	  dtv_type_spec, INTENT(IN) :: dtv
	  INTEGER, INTENT(OUT) :: iostat
	  CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
	 END SUBROUTINE
	``

Note: The actual specific procedure names (the `my ... routine ...` procedure names above) are not significant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred by the names given above; the names are, however, arbitrary.

The following are the characteristics of the arguments:

dtv

> If the parent data transfer statement is a **READ** statement, *dtv* is the argument associated with the effective list item that caused the user-defined derived-type input procedure to be invoked, as if the effective list item were an actual argument in this procedure reference. If the parent data transfer statement is a **WRITE** or **PRINT** statement, *dtv* contains the effective list item.

unit   When you invoke a user-defined derived-type input/output procedure, *unit* has a value as follows:

- If the parent data transfer statement uses a *file-unit-number*, the value is that of the *file-unit-number*.
- If the parent data transfer statement is a **WRITE** statement with an asterisk unit or a **PRINT** statement, the value is the same as that of the OUTPUT_UNIT named constant of the ISO_FORTRAN_ENV intrinsic module.
- If the parent data transfer statement is a **READ** statement with an asterisk unit or a **READ** statement without an *io-control-spec-list*, the value is the same as that of the INPUT_UNIT named constant of the ISO_FORTRAN_ENV intrinsic module.
- Otherwise the parent data transfer statement must access an internal file. In this case the value is negative.

iotype   For formatted data transfer, *iotype* has a value as follows:

- **"LISTDIRECTED"** if the parent data transfer statement specified list directed formatting

- **"NAMELIST"** if the parent data transfer statement specified namelist formatting
- **"DT"** concatenated with the *char-literal-constant*, if any, of the edit descriptor, if the parent data transfer statement contained a format specification and the list item's corresponding edit descriptor was a **DT** edit descriptor.

v_list   For formatted data transfer, *v_list* has values as follows:
- If the *v-list* of the edit descriptor appears in the parent data transfer statement, *v_list* contains the values specified in *v-list*, with the same number of elements, in the same order.
- If there is no *v-list* in the edit descriptor or if the data transfer statement specifies list-directed or namelist formatting, *v_ list* is a zero-sized array.

iostat   is used to report whether an error, end-of-record, or end-of-file condition occurs. Values are assigned to *iostat* as follows:
- If an error condition occurs, the value is positive.
- If an end-of-file condition occurs, the value is that of the named constant IOSTAT_END.
- If an end-of-record condition occurs, the value is that of the named constant IOSTAT_EOR.
- Otherwise, the value is zero.

iomsg   If the *iostat* argument returns a nonzero value, the procedure returns an explanatory message in *iomsg*. Otherwise, the procedure does not change the value of the *iomsg* argument.

Abstract interface (Fortran 2003)

An abstract interface allows you to specify procedure characteristics and dummy argument names without declaring a procedure with those characteristics. You can use an abstract interface to declare interfaces for procedures and deferred bindings. The procedure names defined in an abstract interface block do not have an **EXTERNAL** attribute.

```
►►──ABSTRACT_INTERFACE_statement──────────────────────────────────────────►◄

►►──┬────────────────────────────────┬──────────────────────────────────────►◄
    │  ┌──────────────────────────┐  │
    └─►┼──FUNCTION_interface_body───┼──┘
       ├──SUBROUTINE_interface_body─┤
       └──PROCEDURE_statement───────┘

►►──END_ABSTRACT_INTERFACE_statement──────────────────────────────────────►◄
```
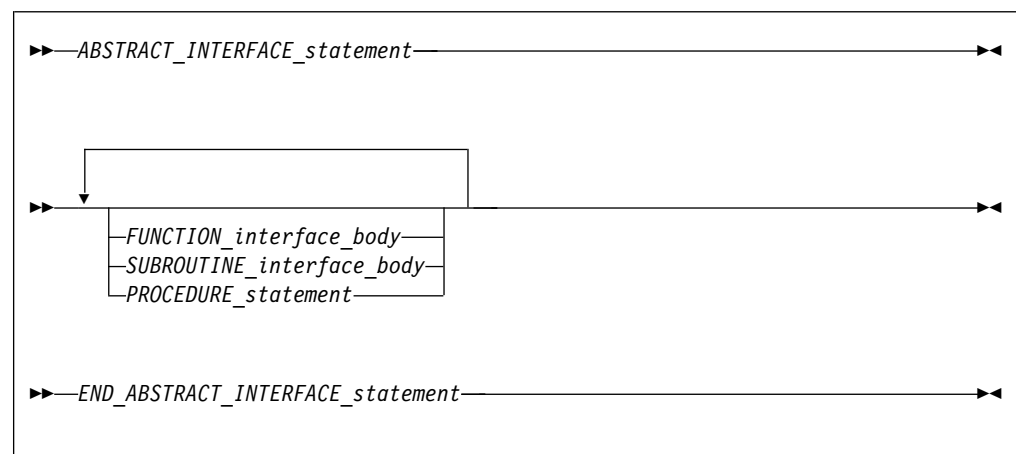
ABSTRACT_INTERFACE_statement
    See "ABSTRACT INTERFACE(Fortran 2003)" on page 285 for syntax details

FUNCTION_interface_body
   See "Interface blocks" on page 164 for syntax details

SUBROUTINE_interface_body
    See "Interface blocks" on page 164 for syntax details

PROCEDURE_statement
    See "PROCEDURE" on page 433 for syntax details

END_ABSTRACT_INTERFACE_statement
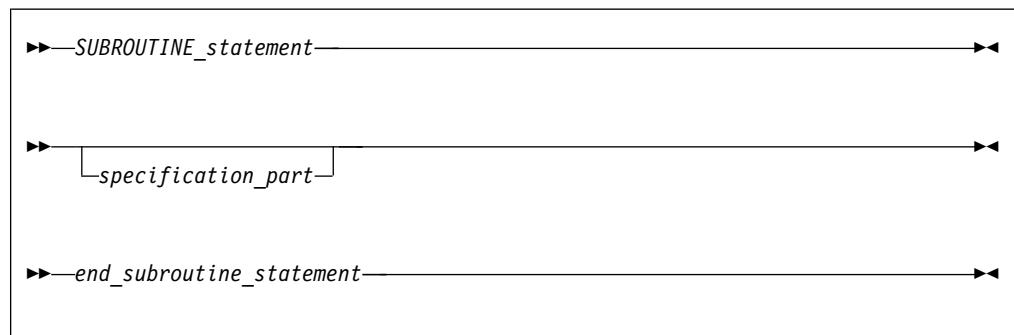   See "END INTERFACE" on page 354 for syntax details

Examples

```
MODULE M
 ABSTRACT INTERFACE
  SUBROUTINE SUB(X,Y)
   INTEGER ,INTENT(IN)::X
   INTEGER ,INTENT(IN)::Y
  END SUBROUTINE
 END INTERFACE
END MODULE

PROGRAM MAIN
 USE M
 PROCEDURE (SUB) SUB1
 PROCEDURE (SUB), POINTER::P
 P=>SUB1
 CALL P(5,10)
END PROGRAM

SUBROUTINE SUB1 (X,Y)
  INTEGER ,INTENT(IN)::X
  INTEGER ,INTENT(IN)::Y
  PRINT*, "The sum of X and Y is: ", X + Y
END SUBROUTINE
```

Related information

For more information see:
- PROCEDURE declaration "PROCEDURE declaration (Fortran 2003)" on page 435
- external procedures "Program units, procedures, and subprograms" on page 161
- dummy procedures "Procedures as dummy arguments" on page 205
- module procedures "Modules" on page 177

Main program

A main program is the program unit that receives control from the system when the executable program is invoked at run time.

```
>>--+-------------------+-------------------------------------------><
     '-PROGRAM_statement-'

>>--+-------------------+-------------------------------------------><
     '-specification_part-'

>>--+----------------+----------------------------------------------><
     '-execution_part-'

>>--+-------------------------+-------------------------------------><
     '-internal_subprogram_part-'

>>----END_PROGRAM_statement-----------------------------------------><
```

PROGRAM_statement
> See "PROGRAM" on page 437 for syntax details

specification_part
> is a sequence of statements from the statement groups numbered **2** , **4** , and **5** in "Order of statements and execution sequence" on page 16

execution_part
> is a sequence of statements from the statement groups numbered **4** and **6** in "Order of statements and execution sequence" on page 16, and which must begin with a statement from statement group **6**

internal_subprogram_part
> See "Internal procedures" on page 162 for details

END_PROGRAM_statement
> See "END" on page 350 for syntax details

A main program cannot contain an **ENTRY** statement, nor can it specify an automatic object.

IBM A **RETURN** statement can appear in a main program. The execution of a **RETURN** statement has the same effect as the execution of an **END** statement. IBM

Modules

A module contains specifications and definitions that can be accessed from other program units. These definitions include data object definitions, namelist groups, derived-type definitions, procedure interface blocks and procedure definitions. F2003

There are two types of modules, intrinsic and nonintrinsic. XL Fortran provides intrinsic modules, while nonintrinsic modules are user-defined.

An intrinsic module can have the same name as other global entities, such as program units, common blocks, external procedures, critical sections, or binding

labels of global entities. A scoping unit must not access both an intrinsic module and a non-intrinsic module with the same name. F2003

IBM Modules define global data, which, like **COMMON** data, is shared across threads and is therefore thread-unsafe. To make an application thread-safe, you must declare the global data as **THREADPRIVATE** or **THREADLOCAL**. See "COMMON" on page 315, **THREADLOCAL**, and **THREADPRIVATE** in the *XL Fortran Optimization and Programming Guide* for more information. IBM

Syntax

MODULE

▶▶──*MODULE_statement*──▶◀

▶▶──▶◀
　　└─*specification_part*─┘

▶▶──▶◀
　　└─*module_subprogram_part*─┘

▶▶──*END_MODULE_statement*──▶◀

module_subprogram_part

▶▶──*CONTAINS_statement*──▶◀

▶▶──┬─*module_subprogram*─┬──▶◀
　　└──────────────────────┘
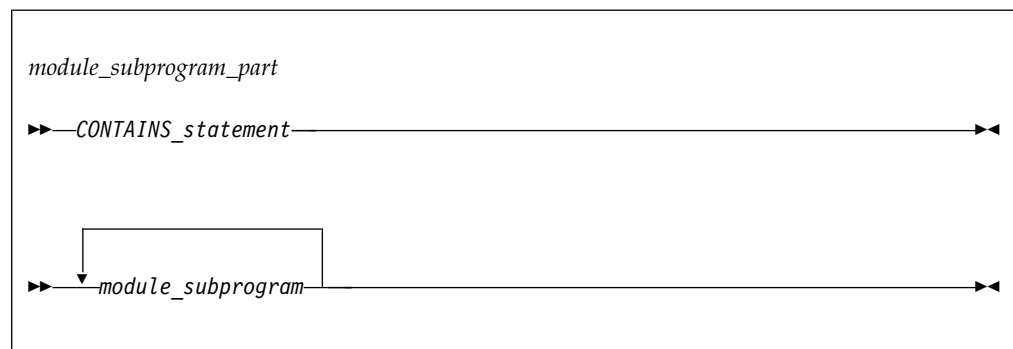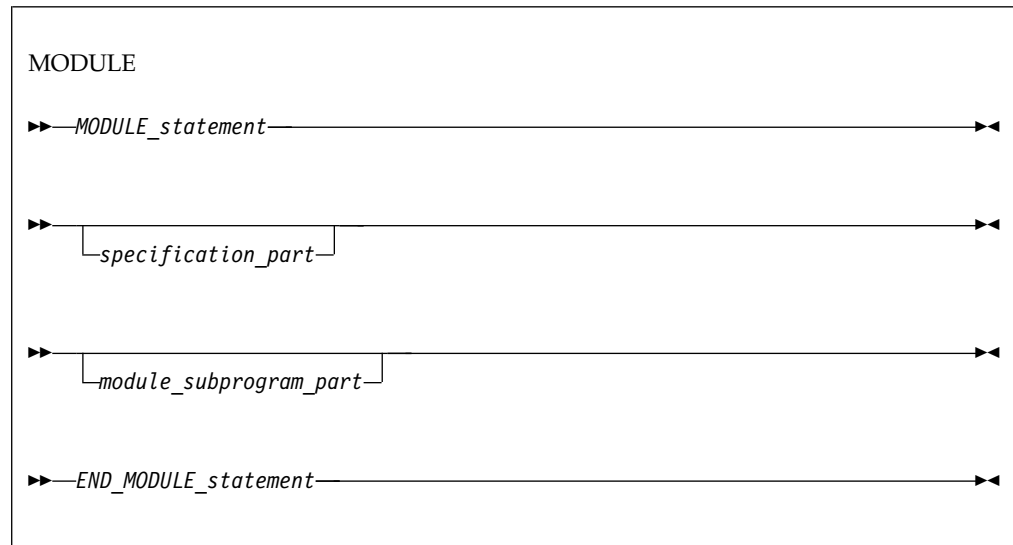
MODULE_statement
> See "MODULE" on page 412 for syntax details.

specification_part
> It is a sequence of statements from the statement groups numbered **2** , **4** , and **5** in "Order of statements and execution sequence" on page 16.

CONTAINS_statement
> See "CONTAINS" on page 322 for syntax details.

END_MODULE_statement
> See "END" on page 350 for syntax details.

Rules

A module subprogram is contained in a module but is not an internal subprogram. Module subprograms must follow a **CONTAINS** statement, and can contain internal procedures. A module procedure is defined by a module subprogram or an entry in a module subprogram.

Executable statements within a module can only be specified in module subprograms.

The declaration of a module function name of type character cannot have an asterisk as a length specification.

specification_part cannot contain statement function statements, **ENTRY** statements, or **FORMAT** statements, although these statements can appear in the specification part of a module subprogram.

Automatic objects and objects with the **AUTOMATIC** attribute cannot appear in the scope of a module.

An accessible module procedure can be invoked by another subprogram in the module or by any scoping unit outside the module through use association (that is, by using the **USE** statement). See "USE" on page 483 for details.

`IBM`

Integer pointers cannot appear in *specification_part* if the pointee specifies a dimension declarator with nonconstant bounds.

All objects in the scope of a module retain their association status, allocation status, definition status, and value when any procedure that accesses the module through use association executes a **RETURN** or **END** statement. See point 4 under "Events causing undefinition" on page 24 for more information.

`IBM`

A module is a host to any module procedures, interface blocks, derived-type definitions it contains, `F2008` or its descendant submodules `F2008` . Through host association, the module procedures, interface blocks, derived-type definitions, `F2008` and descendant submodules `F2008` can access entities in the scope of the module.

A module procedure can be used as an actual argument associated with a dummy procedure argument.

The name of a module procedure is local to the scope of the module and cannot be the same as the name of any entity in the module, except for a common block name.

`F2008` In *specification_part*, you can declare a module procedure interface body to specify the interface of a separate module procedure. See Example 2. You can later define the separate module procedure in one of the descendant submodules. `F2008`

Migration Tips:

- Eliminate common blocks and **INCLUDE** directives
- Use modules to hold global data and procedures to ensure consistency of definitions

FORTRAN 77 source:

```
      COMMON /BLOCK/A, B, C, NAME, NUMBER
      REAL A, B, C
      A = 3
      CALL CALLUP(D)
      PRINT *, NAME, NUMBER
      END
      SUBROUTINE CALLUP (PARM)
        COMMON /BLOCK/A, B, C, NAME, NUMBER
        REAL A, B, C
        ...
        NAME = 3
        NUMBER = 4
      END
```

Fortran 90/95/2003 source:

```
      MODULE FUNCS
        REAL A, B, C            ! Common block no longer needed
        INTEGER NAME, NUMBER    ! Global data
        CONTAINS
           SUBROUTINE CALLUP (PARM)
             ...
             NAME = 3
             NUMBER = 4
           END SUBROUTINE
      END MODULE FUNCS
      PROGRAM MAIN
      USE FUNCS
      A = 3
      CALL CALLUP(D)
      PRINT *, NAME, NUMBER
      END
```

Example 1

```
MODULE m
  INTEGER some_data

  CONTAINS
    SUBROUTINE sub()                    ! Module subprogram
      INTEGER stmtfnc
      stmtfnc(i) = i + 1
      some_data = stmtfnc(5) + inner(3)
      CONTAINS
        INTEGER FUNCTION inner(iarg)    ! Internal subprogram
          inner = iarg * 2
        END FUNCTION
    END SUBROUTINE sub
END MODULE

PROGRAM main
  USE m              ! Main program accesses module m
  CALL sub()
END PROGRAM
```

Example 2 (Fortran 2008)

```
MODULE m
  INTERFACE
    ! A module procedure interface body that specifies the
```

```
      ! interface of a separate module subroutine
    MODULE SUBROUTINE sub(arg)
      INTEGER :: arg
    END SUBROUTINE

    ! A module procedure interface body that specifies the
    ! interface of a separate module function
    INTEGER MODULE FUNCTION func(a, b)
      INTEGER :: a, b
    END FUNCTION
  END INTERFACE
END MODULE
```

Related information

- "Submodules (Fortran 2008)"

Submodules (Fortran 2008)

A submodule is a program unit that extends a module or another submodule. It might provide definitions for procedures whose interfaces are declared in the ancestor module or an ancestor submodule. It might also contain declarations and definitions of other entities. These definitions include data object definitions, namelist groups, derived-type definitions, procedure interface blocks, and procedure definitions.

A submodule cannot have the same name as other global entities, such as program units, common blocks, external procedures, critical sections, or binding labels of global entities. The name of a submodule can be the same as the name of another submodule if they do not have the same ancestor module.

Syntax

SUBMODULE

▶▶──*SUBMODULE_statement*────────────────────────▶◀

▶▶──▶◀
    └─*specification_part*─┘

▶▶──▶◀
    └─*module_subprogram_part*─┘

▶▶──*END_SUBMODULE_statement*────────────────────▶◀

```
 ┌─────────────────────────────────────────────────────────────────────┐
 │                                                                       │
 │  module_subprogram_part                                               │
 │                                                                       │
 │  ►►──CONTAINS_statement───────────────────────────────────────────►◄  │
 │                                                                       │
 │            ┌───────────────────────┐                                  │
 │            │                       │                                  │
 │  ►►────────▼─module_subprogram─────┴──────────────────────────────►◄  │
 │                                                                       │
 └─────────────────────────────────────────────────────────────────────┘
```
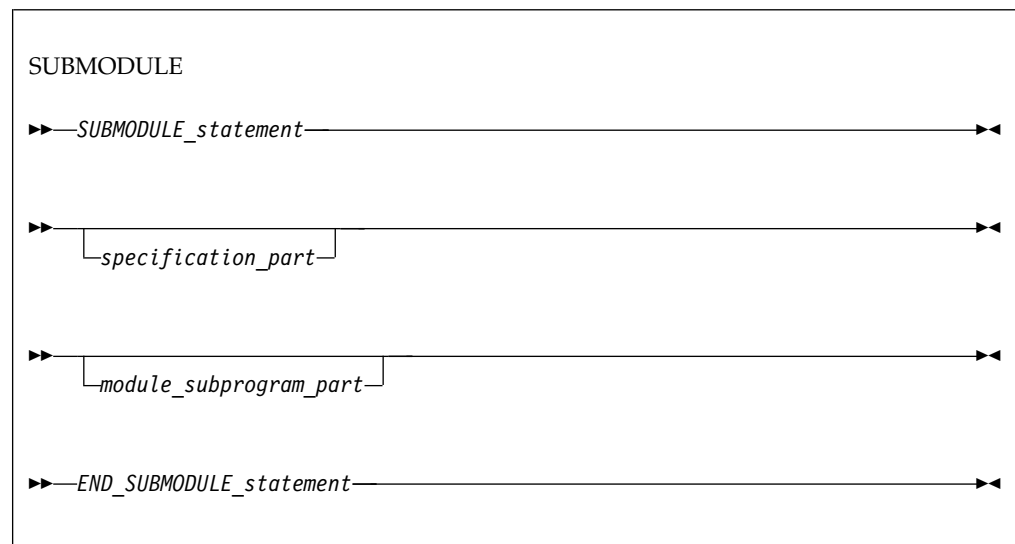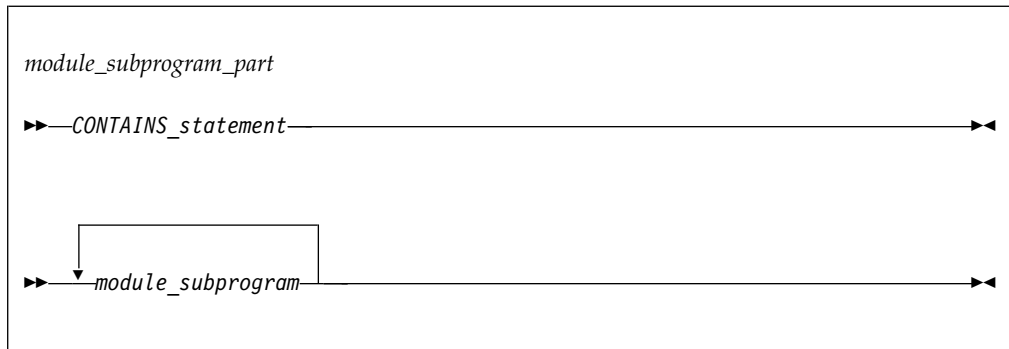
SUBMODULE_statement
>   For syntax details, see SUBMODULE.

specification_part
>   It is a sequence of statements from the statement groups numbered **2** ,
>   **4** , and **5** in "Order of statements and execution sequence" on page 16,
>   excluding **ENTRY** and **FORMAT** statements.

CONTAINS_statement
>   For syntax details, see "CONTAINS" on page 322.

END_SUBMODULE_statement
>   For syntax details, see "END" on page 350.

Rules

A submodule has only one ancestor module and can have one or more ancestor
submodules.

A submodule might contain module subprograms and is the host scoping unit of
the module subprograms. Module subprograms must follow a **CONTAINS**
statement and can contain internal procedures. A module procedure is defined by a
module subprogram or an entry in a module subprogram.

In a submodule, you can define separate module procedures that are declared by
corresponding module procedure interface bodies in the ancestor module or an
ancestor submodule.

A submodule has access to the entities that are declared and defined in its ancestor
module or submodules by host association. A submodule is a host to any module
procedures, interface blocks, or derived-type definitions that it contains. Through
host association, the module procedures, interface blocks, and derived-type
definitions that a submodule contains can access entities in the scope of the
submodule. The declarations and definitions of entities in a submodule are also
accessible by host association in its descendant submodules.

A submodule cannot be accessed by use association.

A local variable of a submodule is accessible only in that submodule scoping unit,
in any contained scoping units, or in its descendant submodules.

A variable, common block, or procedure pointer that is declared in the scoping
unit of a submodule implicitly has the **SAVE** attribute, which might be confirmed
by explicit specification.

The following rules apply to the *specification_part* of a submodule:

- It cannot contain statement function statements, **ENTRY** statements, or **FORMAT** statements, although these statements can appear in the specification part of a module subprogram.
- If a specification expression or constant expression includes a reference to a generic entity, that generic entity cannot have specific procedures that are defined in the submodule subsequent to the specification expression or constant expression.

Example

In this example, two separate module procedures are defined in the submodule n by a subroutine subprogram with the **MODULE** prefix specifier and by a separate module subprogram. Their corresponding module procedure interface bodies are declared in the ancestor module m.

```
MODULE m                              ! The ancestor module m
  INTEGER :: i

  INTERFACE
    MODULE SUBROUTINE sub1(arg1)  ! Module procedure interface body for sub1
      INTEGER :: arg1
    END SUBROUTINE

    MODULE SUBROUTINE sub2(arg2)  ! Module procedure interface body for sub2
      INTEGER :: arg2
    END SUBROUTINE
  END INTERFACE
END MODULE

SUBMODULE (m) n                       ! The descendant submodule n
  INTEGER :: j                        ! Specification part

  CONTAINS                            ! Module subprogram part
    MODULE SUBROUTINE sub1(arg1)  ! Definition of sub1 by subroutine subprogram
      INTEGER :: arg 1
      arg1 = 1
      i = 2                           ! Host association
      j = 3                           ! Host association
    END SUBROUTINE

    MODULE PROCEDURE sub2             ! Definition of sub2 by separate module subprogram
      arg2 = 1
    END PROCEDURE
END SUBMODULE
```

Related information

- "Modules" on page 177
- "SUBMODULE (Fortran 2008)" on page 466
- "Separate module subprograms (Fortran 2008)" on page 189
- "Separate module procedures (Fortran 2008)" on page 188

Module subprograms

A module subprogram is a subprogram that is contained in a module or `F2008` submodule `F2008`, but is not an internal subprogram. A module subprogram has one of the following forms:

- A function subprogram
- A subroutine subprogram

- **F2008** A separate module subprogram **F2008**

Function and subroutine subprograms

A function or subroutine subprogram is a subprogram that has a **FUNCTION** or **SUBROUTINE** statement as its first statement. You can also specify a function in a statement function statement. An external function or subroutine subprogram is a program unit.

Syntax

```
►►──subprogram_statement──────────────────────────────────────►◄


►►──────────────────────────────────────────────────────────►◄
       └─specification_part─┘


►►──────────────────────────────────────────────────────────►◄
       └─execution_part─┘


►►──────────────────────────────────────────────────────────►◄
       └─internal_subprogram_part─┘


►►──end_subprogram_statement──────────────────────────────────►◄
```
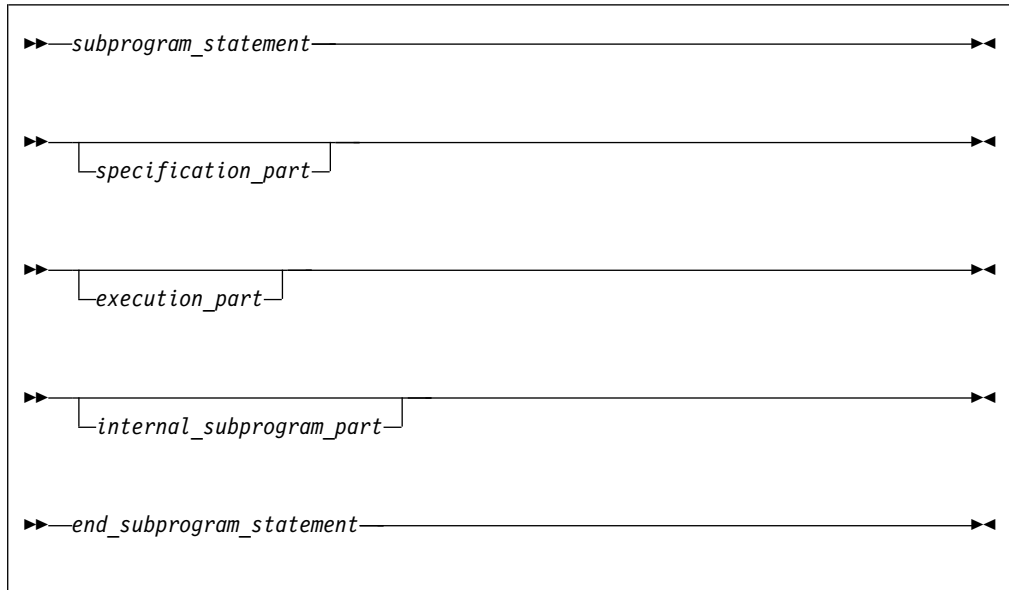
subprogram_statement
    See "FUNCTION" on page 379 or "SUBROUTINE" on page 468 for syntax details

specification_part
    is a sequence of statements from the statement groups numbered **2** , **4** and **5** in "Order of statements and execution sequence" on page 16

execution_part
    is a sequence of statements from the statement groups numbered **4** and **6** in "Order of statements and execution sequence" on page 16, and which must begin with a statement from statement group **6**

internal_subprogram_part
    See "Internal procedures" on page 162 for details

end_subprogram_statement
    See "END" on page 350 for syntax details on the **END** statement for functions and subroutines

Rules

An internal subprogram is declared *after* the **CONTAINS** statement in the main program, a module subprogram, or an external subprogram, but *before* the **END** statement of the host program. The name of an internal subprogram must not be defined in the specification section in the host scoping unit.

An external procedure has global scope with respect to the executable program. In the calling program unit, you can specify the interface to an external procedure in an interface block or you can define the external procedure name with the **EXTERNAL** attribute.

A subprogram can contain any statement except **PROGRAM**, **BLOCK DATA**, **MODULE**, and F2008 **SUBMODULE** F2008 statements. An internal subprogram cannot contain an **ENTRY** statement or an internal subprogram.

Declaring procedures

An EXTERNAL statement, PROCEDURE declaration statement, or a procedure component definition statement can be used to declare a procedure.

An EXTERNAL statement declares external procedures and dummy procedures. See "EXTERNAL" on page 368.

A PROCEDURE declaration statement declares procedure pointers, dummy procedures, and external procedures. For further information on the PROCEDURE declaration statement, see "PROCEDURE declaration (Fortran 2003)" on page 435.

A procedure component definition statement declares procedure pointer components of a derived type definition. See "Procedure pointer components" on page 53.

Procedure pointers (Fortran 2003): A procedure pointer is a procedure that has the EXTERNAL and POINTER attribute. A derived type component which has been declared with the PROCEDURE statement can be a procedure pointer.

A procedure pointer points at a procedure rather than a data object. A procedure pointer can be associated in the same way as a dummy procedure with an external procedure, a module procedure, an intrinsic procedure, or a dummy procedure that is not a procedure pointer. F2008 A procedure pointer can also be associated with an internal procedure. However, an internal procedure cannot be invoked using a procedure pointer after the host instance of the internal procedure completes its execution. F2008 Procedure pointers can have both an explicit and implicit interface, can be structure components and can be associated using procedure pointer assignment.

A dummy procedure with the pointer attribute is a dummy procedure pointer and its associated actual argument is a procedure pointer.

A procedure pointer shall be storage associated only with another procedure pointer; either both interfaces shall be explicit (the characteristics are the same) or both interfaces shall be implicit (both interfaces will be functions or subroutines with the same type and type parameters).

Although both type-bound procedures and procedure pointer components are invoked through an object , the type-bound procedure which is executed depends upon the type of the invoking object whereas procedure pointer components depend upon the value. The **PASS** attribute defines the passed-object dummy arguments of the procedure pointer component

Examples
```
PROCEDURE(PROC), POINTER :: PTR
```

Procedure references

There are two types of procedure references:

- A subroutine is invoked by any of the following cases:
  - execution of a **CALL** statement
  - execution of a defined assignment statement
  - F2003 user-defined derived-type input/output F2003
  - F2003 execution of finalization F2003
- A function is invoked during evaluation of a function reference or defined operation.

Function reference: A function reference is used as a primary in an expression:

```
                                          (1)
►►─┬─ data_ref ─ separator ─ binding_name ──────┬────────────────────►
   │                      (2)                    │
   ├─ procedure_component_ref ──────────────────┤
   └─ function_name ────────────────────────────┘

►─(─┬──────────────────────┬─)──────────────────────────────────────►◄
    └─ actual_argument_spec_list ─┘


Notes:

1    Fortran 2003

2    Fortran 2003
```
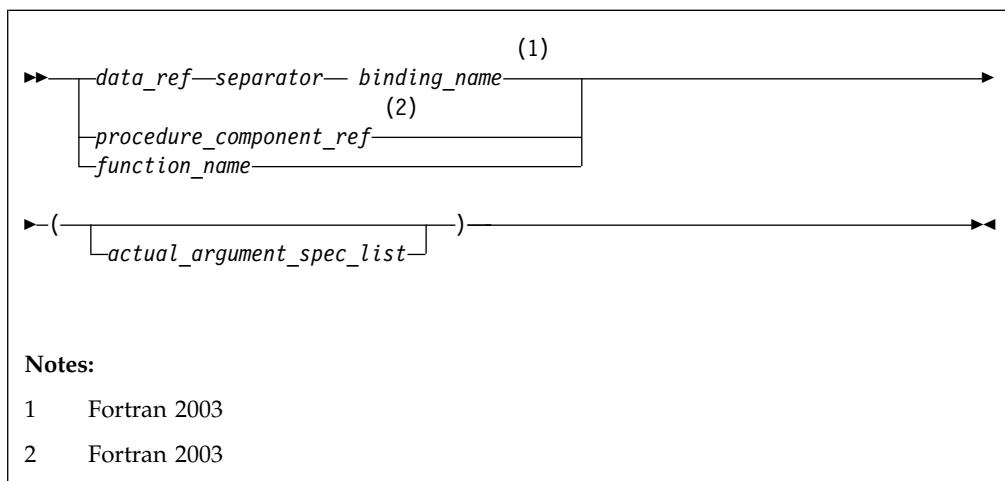
Executing a function reference results in the following order of events:

1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. Control transfers to the specified function.
4. The function is executed.
5. The value (or status or target, for pointer functions) of the function result variable is available to the referencing expression.

Fortran 2003

If the *binding_name* in a function reference is that of a specific function, the function referenced is the one identified by the binding with that name in the dynamic type of the *data-ref*.

If the *binding_name* in a function reference is that of a generic procedure, the generic binding with that name in the declared type of the *data-ref* is used to select a specific binding:

1. If the reference is consistent with one of the specific bindings of that generic binding, that specific binding is selected.
2. Otherwise, if the reference is consistent with an elemental reference to one of the specific bindings of that generic binding, that specific binding is selected.

The reference is to the procedure identified by the binding with the same name as the selected specific binding, in the dynamic type of the *data-ref*.

─────────────────── **End of Fortran 2003** ───────────────────

Execution of a function reference must not alter the value of any other data item within the statement in which the function reference appears. Invocation of a function reference in the logical expression of a logical **IF** statement or **WHERE** statement can affect entities in the statement that is executed when the value of the expression is true.

─────────────────── **IBM extension** ───────────────────

The argument list built-in functions **%VAL** and **%REF** are supplied to aid interlanguage calls by allowing arguments to be passed by value and by reference, respectively. They can be specified in non-Fortran procedure references and in a subprogram statement in an interface body. (See "%VAL and %REF (IBM extension)" on page 196.) See Statement Function and Recursion examples of function references.

─────────────────── **End of IBM extension** ───────────────────

On entry to an allocatable function, the allocation status of the result variable becomes not allocated

The function result variable may be allocated and deallocated any number of times during the execution of the function. However, it shall be allocated and have a defined value on exit from the function. Automatic deallocation of the result variable does not occur immediately on exit from the function, but instead occurs after execution of the statement in which the function reference occurs.

Examples of subprograms and procedure references:

```
MODULE QUAD_MOD
   TYPE QUAD_TYPE
      REAL:: a, b, c
      CONTAINS
      PROCEDURE Q2
   END TYPE

   INTERFACE
      SUBROUTINE Q2(T,QUAD) ! External subroutine
         IMPORT QUAD_TYPE
         CLASS(QUAD_TYPE) T
         REAL QUAD
      END SUBROUTINE
   END INTERFACE
END MODULE
PROGRAM MAIN
   USE QUAD_MOD
   REAL QUAD,X2,X1,X0,A,C3
   TYPE(QUAD_TYPE) QT
   QUAD=0; A=X1*X2
   X2 = 2.0
   X1 = SIN(4.5) ! Reference to intrinsic function
   X0 = 1.0
   QT = QUAD_TYPE(X2, X1, X0)
   CALL Q(X2,X1,X0,QUAD) ! Reference to external subroutine
   CALL QT%Q2(QUAD) ! Reference to a  subroutine
   C3 = CUBE() ! Reference to internal function
   CONTAINS
   REAL FUNCTION CUBE() ! Internal function
```

```
      CUBE = A**3
   END FUNCTION CUBE
END
SUBROUTINE Q(A,B,C,QUAD) ! External subroutine
   REAL A,B,C,QUAD
   QUAD = (-B + SQRT(B**2-4*A*C)) / (2*A)
END SUBROUTINE Q
SUBROUTINE Q2(T,QUAD) ! External subroutine
   USE QUAD_MOD
   TYPE(QUAD_TYPE) T
   REAL QUAD
   QUAD = (-T%B + SQRT(T%B**2-4*T%A*T%C)) / (2*T%A)
END SUBROUTINE Q2
```

Examples of allocatable function results:

```
FUNCTION INQUIRE_FILES_OPEN() RESULT(OPENED_STATUS)
  LOGICAL,ALLOCATABLE :: OPENED_STATUS(:)
  INTEGER I,J
  LOGICAL TEST
  DO I=1000,0,-1
    INQUIRE(UNIT=I,OPENED=TEST,ERR=100)
    IF (TEST) EXIT
100 CONTINUE
  END DO
  ALLOCATE(OPENED_STATUS(0:I))
  DO J=0,I
    INQUIRE(UNIT=J,OPENED=OPENED_STATUS(J))
  END DO
END FUNCTION INQUIRE_FILES_OPEN
```

Separate module procedures (Fortran 2008)

A separate module procedure is a module procedure that is defined by a separate
module subprogram, or by a function or subroutine subprogram whose initial
statement contains the **MODULE** prefix specifier. A separate module procedure
defines a module procedure interface body.

- A separate module procedure must be previously declared by a module
  procedure interface body in the containing program unit, in the ancestor
  module, or in an ancestor submodule.
- A separate module procedure cannot be defined more than once.
- A separate module procedure must have the same characteristics and dummy
  argument names as its corresponding module procedure interface body.
- If you specify a binding label for a separate module procedure, the binding label
  must be the same as that is specified in the corresponding module procedure
  interface body.
- You must specify the **RECURSIVE** prefix specifier for a separate module
  procedure if and only if it is also specified for the corresponding module
  procedure interface body.
- When a separate module procedure is a function, the following rules apply to
  the result variable:
  - If it is defined by a separate module subprogram, the result variable name is
    determined by the **FUNCTION** statement in the corresponding module
    procedure interface body.
  - If it is defined by a function subprogram with the **MODULE** prefix specifier,
    the result variable name is determined by the **FUNCTION** statement of the
    subprogram.

For how the **MODULE** prefix specifier is used for a function or subroutine
subprogram, see "FUNCTION" on page 379 or "SUBROUTINE" on page 468.

Related information
- "Modules" on page 177
- "Submodules (Fortran 2008)" on page 181
- "Separate module subprograms (Fortran 2008)"

Separate module subprograms (Fortran 2008)

A separate module subprogram defines a separate module procedure that is declared by a corresponding module procedure interface body.
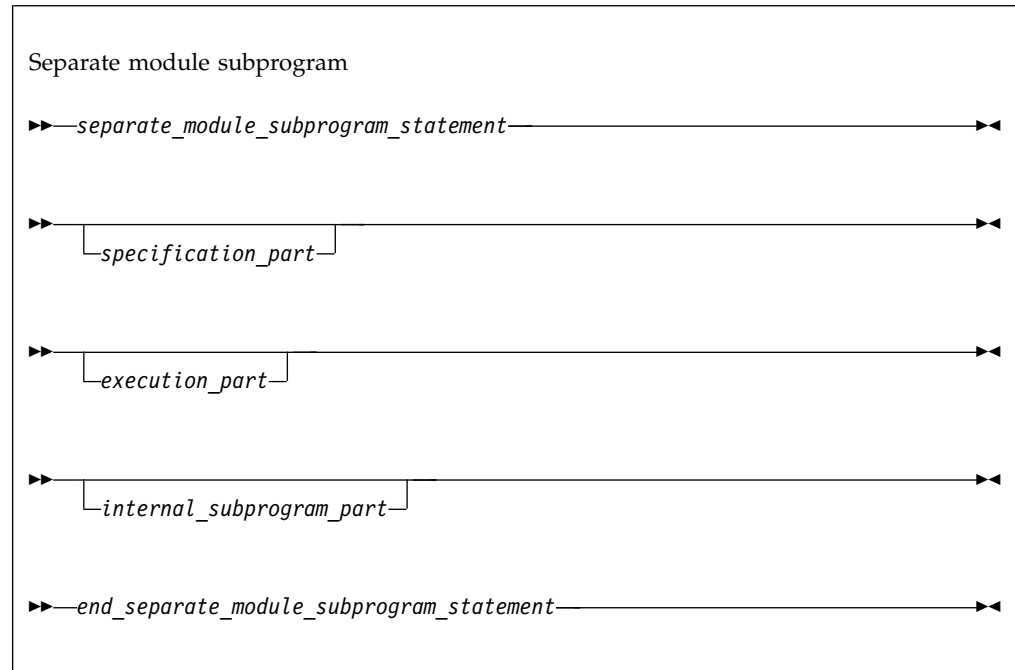
Syntax

separate_module_subprogram_statement
> For syntax details, see MODULE PROCEDURE.

specification_part
> It is a sequence of statements from the statement groups numbered **2**, **4** and **5** in "Order of statements and execution sequence" on page 16.

execution_part
> It is a sequence of statements from the statement groups numbered **4** and **6** in "Order of statements and execution sequence" on page 16, which must begin with a statement from statement group **6**.

internal_subprogram_part
> For details, see "Internal procedures" on page 162.

end_separate_module_subprogram_statement
> For syntax details, see "END" on page 350.

Rules

The interface of a separate module subprogram must be declared by a module procedure interface body in one of the following places:

- The *specification_part* of the program unit where the separate module procedure is defined.
- The ancestor module.
- An ancestor submodule.

If a separate module procedure that is defined by a separate module subprogram is a function, the variable name of the function result is determined by the **FUNCTION** statement in the corresponding module procedure interface body.

Example

This example shows how a separate module procedure is defined by a separate module subprogram in a submodule.

```
MODULE m
  INTEGER :: i

  INTERFACE
    MODULE SUBROUTINE sub(arg)
      INTEGER :: arg
    END SUBROUTINE
  END INTERFACE
END MODULE

SUBMODULE (m) n
  INTEGER :: j

  CONTAINS
    MODULE PROCEDURE sub        ! Separate module subprogram
      arg = 1
      i = 2
      j = 3
    END PROCEDURE
END SUBMODULE
```

Related information
- "Modules" on page 177
- "Submodules (Fortran 2008)" on page 181
- "Separate module procedures (Fortran 2008)" on page 188

Block data program unit

A block data program unit provides initial values for objects in named common blocks.

```
►►──BLOCK_DATA_statement──────────────────────────────────────────►◄


►►─────────────────────────────────────────────────────────────────►◄
     └─specification_part─┘


►►──END_BLOCK_DATA_statement──────────────────────────────────────►◄
```
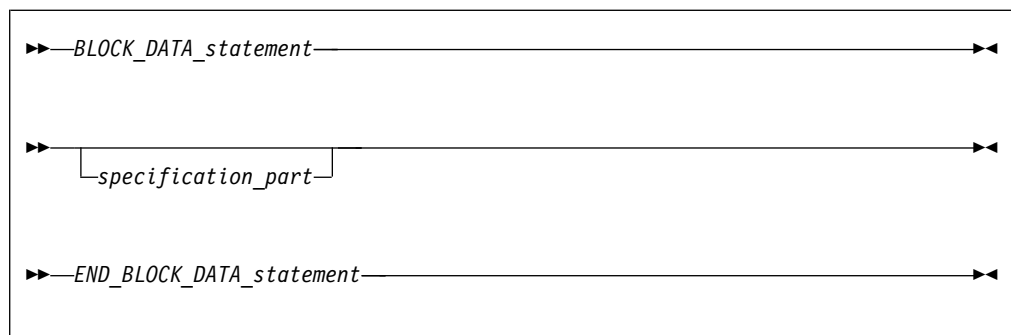
BLOCK_DATA_statement
      See "BLOCK DATA" on page 299 for syntax details

specification_part
> is a sequence of statements from the statement groups numbered **2** , **4** , and **5** in "Order of statements and execution sequence" on page 16

END_BLOCK_DATA_statement
> See "END" on page 350 for syntax details

In *specification_part*, you can specify type declaration,**USE**, **IMPLICIT**, **COMMON**, **DATA**, **EQUIVALENCE**, and integer pointer statements, derived-type definitions, and the allowable attribute specification statements. The only attributes that can be specified are: F2003 **BIND** F2003 , **DIMENSION**, **INTRINSIC**, **PARAMETER**, **POINTER**, **SAVE**, **TARGET**, and F2003 **ASYNCHRONOUS** F2003 .

A type declaration statement in a block data *specification-part* must not contain **ALLOCATABLE** or **EXTERNAL** attribute specifiers.

You can have more than one block data program unit in an executable program, but only one can be unnamed. You can also initialize multiple named common blocks in a block data program unit.

Restrictions on common blocks in block data program units are:

- All items in a named common block must appear in the **COMMON** statement, even if they are not all initialized.
- The same named common block must not be referenced in two different block data program units.
- Only nonpointer objects in named common blocks can be initialized in block data program units.
- Objects in blank common blocks cannot be initialized.

Examples

```
PROGRAM MAIN
  COMMON /L3/ C, X(10)
  COMMON /L4/ Y(5)
END PROGRAM
BLOCK DATA BDATA
  COMMON /L3/ C, X(10)
  DATA C, X /1.0, 10*2.0/   ! Initializing common block L3
END BLOCK DATA

BLOCK DATA                  ! An unnamed block data program unit
  PARAMETER (Z=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*Z/
END BLOCK DATA
```

Intrinsic procedures

An intrinsic procedure is a procedure already defined by XL Fortran. See Chapter 14, "Intrinsic procedures," on page 567 for details.

You can reference some intrinsic procedures by a generic name, some by a specific name, and some by both:

A generic intrinsic function
> does not require a specific argument type and usually produces a result of the same type as that of the argument, with some exceptions. Generic names simplify references to intrinsic procedures because the same

procedure name can be used with more than one type of argument; the type and kind type parameter of the arguments determine which specific function is used.

A specific intrinsic function
requires a specific argument type and produces a result of a specific type.

A specific intrinsic function name can be passed as an actual argument. If a specific intrinsic function has the same name as a generic intrinsic function, the specific name is referenced. All references to a dummy procedure that are associated with a specific intrinsic procedure must use arguments that are consistent with the interface of the intrinsic procedure. Specific intrinsic functions may be procedure pointer targets.

Whether or not you can pass the name of an intrinsic procedure as an argument depends on the procedure. You can use the specific name of an intrinsic procedure that has been specified with the **INTRINSIC** attribute as an actual argument in a procedure reference.

- An **IMPLICIT** statement does not change the type of an intrinsic function.
- If an intrinsic name is specified with the **INTRINSIC** attribute, the name is always recognized as an intrinsic procedure.

Conflicts between intrinsic procedure names and other names

When you declare a data object with the same name as an intrinsic procedure, the intrinsic procedure is inaccessible.

A generic interface block can extend or redefine a generic intrinsic function, as described in "Interface blocks" on page 164. If the function already has the **INTRINSIC** attribute, it is extended; otherwise, it can be redefined.

Arguments

Actual argument specification

Notes:

1   IBM extension
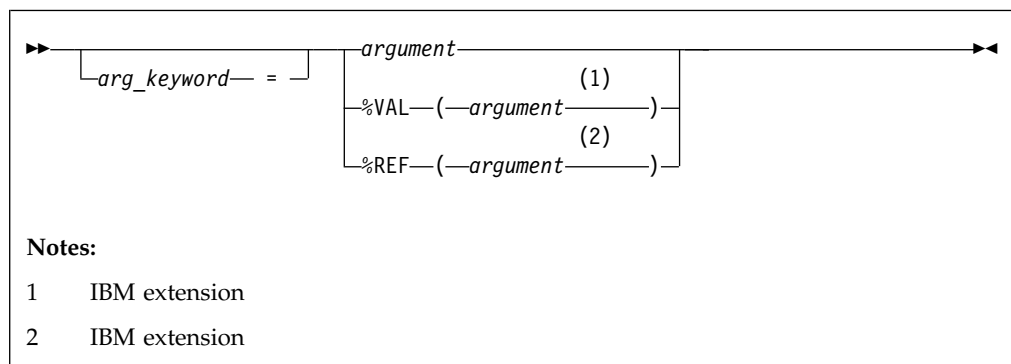
2   IBM extension

arg_keyword
is a dummy argument name in the explicit interface of the procedure being invoked

argument
is an actual argument

> **IBM** **%VAL, %REF**
> specifies the passing method. See "%VAL and %REF (IBM extension)" on page 196 for more information. **IBM**

An actual argument appears in the argument list of a procedure reference. The following actual arguments are valid in procedure references:

- An expression
- A variable
- A procedure name
- An alternate return specifier (if the actual argument is in a **CALL** statement), having the form *stmt_label*, where *stmt_label* is the statement label of a branch target statement in the same scoping unit as the **CALL** statement.

An actual argument specified in a statement function reference must be a scalar object.

In Fortran 2003, a procedure name cannot be the name of an internal procedure, statement function, or the generic name of a procedure, unless it is also a specific name. **F2008** However, Fortran 2008 permits the name of an internal procedure. **F2008**

The rules and restrictions for referencing a procedure described in "Procedure references" on page 186. You cannot use a non-intrinsic elemental procedure as an actual argument in Fortran 95.

Argument keywords

Argument keywords allow you to specify actual arguments in a different order than the dummy arguments. With argument keywords, any actual arguments that correspond to optional dummy arguments can be omitted; that is, dummy arguments that merely serve as placeholders are not necessary.

Each argument keyword must be the name of a dummy argument in the explicit interface of the procedure being referenced. An argument keyword must not appear in an argument list of a procedure that has an implicit interface.

In the argument list, if an actual argument is specified with an argument keyword, the subsequent actual arguments in the list must also be specified with argument keywords.

An argument keyword cannot be specified for label parameters. Label parameters must appear before referencing the argument keywords in that procedure reference.

Examples

```
INTEGER MYARRAY(1:10)
INTERFACE
  SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
    INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
    LOGICAL, OPTIONAL :: DESCENDING
  END SUBROUTINE
END INTERFACE
CALL SORT(MYARRAY, ARRAY_SIZE=10)  ! No actual argument corresponds to the
                                   ! optional dummy argument DESCENDING
END
SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
  INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
  LOGICAL, OPTIONAL :: DESCENDING
```

```
      IF (PRESENT(DESCENDING)) THEN
               .
               .
               .
END SUBROUTINE
```

Dummy arguments

```
►►─┬─dummy_arg_name──────────────────────────────────────────┬─►◄
   │          (1)                                             │
   ├─%VAL──────(──┬─dummy_arg_name─┬──)─┤                     │
   │              └─*──────────────┘                         │
   │          (2)                                             │
   └─%REF──────(──┬─dummy_arg_name─┬──)─┘
                  └─*──────────────┘
```

Notes:

1   IBM extension

2   IBM extension
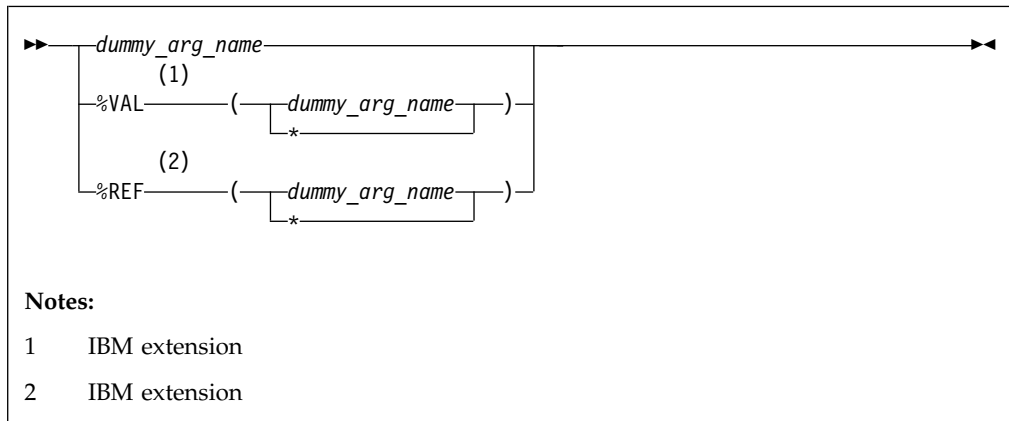
A dummy argument is specified in a Statement Function statement, **FUNCTION**
statement, **SUBROUTINE** statement, or **ENTRY** statement. Dummy arguments in
statement functions, function subprograms, interface bodies, and subroutine
subprograms indicate the types of actual arguments and whether each argument is
a scalar value, array, procedure, or statement label. A dummy argument in an
external, module, or internal subprogram definition, or in an interface body, is
classified as one of the following:

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

─── IBM extension ───

%VAL or **%REF** can only be specified for a dummy argument in a **FUNCTION** or
SUBROUTINE statement in an interface block. The interface must be for a
non-Fortran procedure interface. If **%VAL** or **%REF** appears in an interface block
for an external procedure, this passing method is implied for each reference to that
procedure. If an actual argument in an external procedure reference specifies
%VAL or **%REF**, the same passing method must be specified in the interface block
for the corresponding dummy argument. See "%VAL and %REF (IBM extension)"
on page 196 for more details.

─── End of IBM extension ───

A dummy argument in a statement function definition is classified as a variable
name.

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

The name of a variable that appears as a dummy argument in a statement function
statement has a scope of the statement in which it appears. It has the type that it
would have if it were the name of a variable in the scoping unit that includes the
statement function. It cannot have the same name as an accessible array.

Argument association

Actual arguments are associated with dummy arguments when a function or subroutine is referenced. In a procedure reference, the actual argument list identifies the correspondence between the actual arguments provided in the list and the dummy arguments of the subprogram.

Fortran 2003

The reduced dummy argument list is the full dummy argument list or, if there is a passed-object dummy argument, the dummy argument list with the passed object dummy argument omitted. When there is no argument keyword, an actual argument is associated with the dummy argument that occupies the corresponding position in the reduced dummy argument list. The first actual argument becomes associated with the first dummy argument in the reduced list, the second actual argument with the second dummy argument, continuing until reaching the end of the list. Each actual argument must be associated with a dummy argument.

End of Fortran 2003

When a keyword is present, the actual argument is associated with the dummy argument whose name is the same as the argument keyword. In the scoping unit that contains the procedure reference, the names of the dummy arguments must exist in an accessible explicit interface.

Argument association within a subprogram terminates upon execution of a **RETURN** or **END** statement in the subprogram. There is no retention of argument association between one reference of a subprogram and the next reference of the subprogram, unless you specify **-qxlf77=persistent** and the subprogram contains at least one entry procedure.

If associated with a null argument in a procedure reference, the corresponding dummy argument is undefined and undefinable.

IBM extension

Except when **%VAL** or the **VALUE** attribute is used, 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. If the corresponding actual argument is an expression or an array section with vector subscripts, the calling procedure reserves storage for the actual argument, and the subprogram must not define, redefine, or undefine the dummy argument.

If the actual argument is specified with **%VAL**, or the corresponding dummy argument has the **VALUE** attribute, the subprogram does not have access to the storage area of the actual argument.

End of IBM extension

Actual arguments must agree in type and type parameters with their corresponding dummy arguments (and in shape if the dummy arguments are pointers or assumed-shape), except for two cases: a subroutine name has no type and must be associated with a dummy procedure name that is a subroutine, 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, redefined, or undefined during that subprogram. For example, if a subroutine definition is:

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

and it is referenced by:

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

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 be defined, redefined, or undefined during the execution of subroutine XYZ or by any procedures referenced by XYZ.

If a dummy argument becomes associated with an entity in a common block or an entity accessible through use or host association, the value of the entity must only be altered through the use of the dummy argument name, while the entity is associated with the dummy argument. If any part of a data object is defined through a dummy argument, the data object can be referenced only through that dummy argument, either before or after the definition occurs. These restrictions also apply to pointer targets.

IBM If you have programs that do not conform to these restrictions, using the compiler option **-qalias=nostd** may be appropriate. See the **-qalias** option in the *XL Fortran Compiler Reference* for details. **IBM**

An allocated allocatable component of an actual argument that is associated with an **INTENT(OUT)** dummy argument is deallocated on procedure entry so that the corresponding component of the dummy argument has an allocation status of not allocated. This ensures that any pointers that point to the previous contents of the allocatable component of the variable become undefined.

%VAL and %REF (IBM extension)

To call subprograms written in languages other than Fortran (for example, user-written C programs, or Linux operating system routines), the actual arguments may need to be passed by a method different from the default method used by XL Fortran. The default method passes the address of the actual argument and, if it is of type character, the length. (Use the **-qnullterm** compiler option to ensure that scalar character constant expressions are passed with terminating null strings. See **-qnullterm** option in the *XL Fortran Compiler Reference* for details.)

The default passing method can be changed by using the **%VAL** and **%REF** built-in functions in the argument list of a **CALL** statement or function reference, or with the dummy arguments in interface bodies. These built-in functions specify the way an actual argument is passed to the external subprogram.

%VAL and **%REF** built-in functions cannot be used in the argument lists of Fortran procedure references, nor can they be used with alternate return specifiers.

The argument list built-in functions are:

%VAL This built-in function can be used with actual arguments that are
CHARACTER(1), logical, integer, real, complex expressions, or sequence

derived type. Objects of derived type cannot contain character structure components whose lengths are greater than 1 byte, or arrays.

%VAL cannot be used with actual arguments that are arrays, derived types with allocatable components, procedure names, or character expressions of length greater than 1 byte.

%REF This built-in function causes the actual argument to be passed by reference; that is, only the address of the actual argument is passed. Unlike the default passing method, **%REF** does not pass the length of a character argument. If such a character argument is being passed to a C routine, the string must be terminated with a null character (for example, using the **-qnullterm** option) so that the C routine can determine the length of the string.

Examples

```
EXTERNAL FUNC
CALL RIGHT2(%REF(FUNC))        ! procedure name passed by reference
REAL XVAR
CALL RIGHT3(%VAL(XVAR))        ! real argument passed by value

IVARB=6
CALL TPROG(%VAL(IVARB))        ! integer argument passed by value
```

See "VALUE (Fortran 2003)" on page 487 for a standard-conforming alternative to **%VAL**.

See **Interlanguage calls** in the *XL Fortran Optimization and Programming Guide* for more information.

Intent of dummy arguments

With the **INTENT** attribute, you can explicitly specify the intended use of a dummy argument. Use of this attribute may improve optimization of the program's calling procedure when an explicit interface exists. Also, the explicitness of argument intent may provide more opportunities for error checking. See "INTENT" on page 403 for syntax details.

──────────────── **IBM extension** ────────────────

The following table outlines passing method of XL Fortran for internal procedures (not including assumed-shape, pointer, or allocatable dummy arguments):

Table 22. Passing method and intent

Argument Type	Intent(IN)	Intent(OUT)	Intent(INOUT)	No Intent
Non-CHARACTER Scalar	**VALUE**	default	default	default
CHARACTER*1 Scalar	**VALUE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Scalar	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*(*) Scalar	default	default	default	default
Derived Type [1] Scalar	VALUE	default	default	default

Table 22. Passing method and intent (continued)

Argument Type	Intent(IN)	Intent(OUT)	Intent(INOUT)	No Intent
Derived Type [2] Scalar	default	default	default	default
Non-CHARACTER Array	default	default	default	default
CHARACTER*1 Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*(*) Array	default	default	default	default
Derived Type [3] Array	default	default	default	default

────────── **End of IBM extension** ──────────

Optional dummy arguments

The **OPTIONAL** attribute specifies that a dummy argument need not be associated with an actual argument in a reference to a procedure. Some advantages of the **OPTIONAL** attribute include:

- The use of optional dummy arguments to override default behavior. For an example, see "Argument keywords" on page 193.
- Additional flexibility in procedure references. For example, a procedure could include optional arguments for error handlers or return codes, but you can select which procedure references would supply the corresponding actual arguments.

See "OPTIONAL" on page 423 for details about syntax and rules.

The passed-object dummy argument

┌───────────────────── **Fortran 2003** ─────────────────────┐

In a reference to a procedure that has a passed-object dummy argument, the *data_ref* of the function reference or call statement is associated, as an actual argument, with the passed object dummy argument. See passed-object dummy arguments.

────────── **End of Fortran 2003** ──────────

Restrictions on optional dummy arguments not present

A dummy argument is present in an instance of a subprogram if it is associated with an actual argument that is not a dummy argument itself, or if it is associated with a dummy argument that is either nonoptional or present in the invoking subprogram. A dummy argument that is not optional must be present.

---

1. A data object of derived type with no array components or CHARACTER*n components, where $n > 1$.

2. A data object of derived type with array components or CHARACTER*n components, where $n > 1$.

3. A data object of derived-type with components of any type, size and rank.

A dummy argument or an entity that is host associated with a dummy argument is absent under one of these conditions:

- It does not correspond to an actual argument.
- It corresponds to an actual argument that is not present.

F2008 When the -qxlf2008=checkpresence option is in effect, a dummy argument or an entity that is host associated with a dummy argument is also absent under one of these conditions:

- It does not have the ALLOCATABLE attribute, and corresponds to an actual argument that has the ALLOCATABLE attribute but is not allocated.
- It does not have the POINTER attribute, and corresponds to an actual argument that has the POINTER attribute but is not associated.

Exceptions:

The -qxlf2008=checkpresence option does not affect argument presence under any of these conditions:

- A procedure pointer actual argument is supplied to an optional dummy procedure argument.
- A pointer or allocatable actual argument is supplied to an optional argument of an intrinsic procedure.
- A pointer or allocatable actual argument is supplied to an optional dummy argument of an elemental procedure.

F2008

An optional dummy argument that is not present must conform to the following rules:

- If it is a dummy data object, it must not be referenced or defined. If the dummy data object is of a type for which default initialization can be specified, the initialization has no effect.
- It must not be used as the data_target or proc_target of a pointer assignment
- If it is a procedure or procedure pointer, it must not be invoked.
- It must not be supplied as an actual argument that corresponds to a nonoptional dummy argument, except as the argument of the **PRESENT** intrinsic function.
- A subobject of an optional dummy argument that is not present must not be supplied as an actual argument that corresponds to an optional dummy argument.
- If the optional dummy argument that is not present is an array, it must not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument that corresponds to a nonoptional dummy argument of that elemental procedure.
- If the optional dummy argument that is not present is a pointer, it must not be allocated, deallocated, nullified, pointer-assigned or supplied as an actual argument that corresponds to a nonpointer dummy argument, except as the argument of the **PRESENT** intrinsic function.
- If the optional dummy argument that is not present is allocatable, it must not be allocated, deallocated, or supplied as an actual argument corresponding to a nonallocatable dummy argument other than as the argument of the **PRESENT** intrinsic function.
- If it has length type parameters, they must not be the subject of an inquiry.

- **F2003** An optional dummy argument that is not present must not be used as the *selector* in an **ASSOCIATE** or **SELECT TYPE** construct. **F2003**

Length of character arguments

If the length of a character dummy argument is a nonconstant specification expression or is a colon, the object is a dummy argument with a run-time length. A character dummy argument with a colon length is a deferred length character dummy argument. If an object that is not a dummy argument has a run-time length and is not deferred length, it is an automatic object. See "Automatic objects" on page 20 for details.

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. **%REF** cannot be specified for a character dummy argument with inherited length.

Variables as dummy arguments

F2003 If a dummy argument is neither allocatable nor a pointer, it must be type-compatible with the associated actual argument. If a dummy argument is allocatable or a pointer, the associated actual argument is polymorphic only if the dummy argument is polymorphic, and the declared type of the actual argument is the same as the declared type of the dummy argument. **F2003**

If the actual argument is scalar, the corresponding dummy argument belongs to one of the following categories:
- The dummy argument must be scalar unless the actual argument meets one of the following conditions:
  - **F2003** The actual argument is of type default character. **F2003**
  - **F2003** The actual argument is of type character with the C character kind. **F2003**
  - The actual argument is an element or substring of an element of an array that is not an assumed-shape or pointer array.
- **TS 29113** The dummy argument is an assumed-rank object. **TS 29113**
- **TS 29113** The dummy argument is an assumed-size array of assumed-type. **TS 29113**

If the actual argument is allocatable, the corresponding dummy argument must also be allocatable. If the procedure is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments must agree. A scalar dummy argument can be associated only with a scalar actual argument.

F2003 If the procedure is nonelemental and is referenced by a generic name or as defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments must agree. **F2003**

The following apply to dummy arguments used in elemental subprograms:
- All dummy arguments must be scalar, and cannot have the **F2003** **ALLOCATABLE** or **F2003** **POINTER** attribute.

- A dummy argument, or a suboject thereof, cannot be used in a specification expression, except if it is used as an argument to the **BIT_SIZE**, **KIND**, or **LEN** intrinsic functions, or as an argument to one of the numeric inquiry intrinsic functions, see Chapter 14, "Intrinsic procedures," on page 567.
- A dummy argument cannot be an asterisk.
- A dummy argument cannot be a dummy procedure.

`F2003` If a scalar dummy argument is of type character, its length must be less than or equal to the length of the actual argument. The dummy argument is associated with the leftmost characters of the actual argument. If the character dummy argument is an array, the length restriction applies to the entire array rather than each array element. That is, the lengths of associated array elements can vary, although the whole dummy argument array cannot be longer than the whole actual argument array. `F2003`

If the dummy argument is an assumed-shape array, `F2003` the rank of the actual argument must be the same as the rank of the dummy argument; `F2003` the actual argument must not be an assumed-size array or a scalar, including a designator for an array element or an array element substring.

If the dummy argument is an explicit-shape or assumed-size array, and if the actual argument is a noncharacter array, the size of the dummy argument must not exceed the size of the actual argument array. Each actual array element is associated with the corresponding dummy array element. If the actual argument is a noncharacter array element with a subscript value of `as`, the size of the dummy argument array must not exceed the size of the actual argument array + 1 - `as`. The dummy argument array element with a subscript value of `ds` becomes associated with the actual argument array element that has a subscript value of `as + ds - 1`.

If an actual argument is a character array, character array element, or character substring, 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.

You can define a dummy argument that is a variable name within a subprogram if the associated actual argument is a variable. You must not redefine a dummy argument that is a variable name within a subprogram if the associated actual argument is not definable.

If the actual argument is an array section with a vector subscript, the associated dummy argument cannot be defined and must not have the **INTENT(OUT)**, **INTENT(INOUT)**, **VOLATILE**, or `F2003` **ASYNCHRONOUS** `F2003` attribute.

If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has either the **VOLATILE** or `F2003` **ASYNCHRONOUS** `F2003` attribute, that dummy argument must be an assumed-shape array.

If an actual argument is an nonpointer array with the **VOLATILE** or `F2003` **ASYNCHRONOUS** `F2003` attribute `F2008` but is not simply contiguous, `F2008` and the corresponding dummy argument has either the **VOLATILE** or `F2003` **ASYNCHRONOUS** `F2003` attribute, that dummy argument must be an assumed-shape array `F2008` without the **CONTIGUOUS** attribute. `F2008`

If an actual argument is an array pointer with the **VOLATILE** or F2003 **ASYNCHRONOUS** F2003 attribute F2008 but without the **CONTIGUOUS** attribute F2008 , and the corresponding dummy argument has either the **VOLATILE** or F2003 **ASYNCHRONOUS** F2003 attribute, that dummy argument must be an array pointer or an assumed-shape array F2008 without the **CONTIGUOUS** attribute. F2008

F2008 If the dummy argument is a pointer with the **CONTIGUOUS** attribute, the corresponding actual argument must be simply contiguous. F2008

F2003 Except in references to intrinsic inquiry functions, F2003 if a nonpointer dummy argument is associated with a pointer actual argument, the actual argument must be currently associated with a target, to which the dummy argument becomes argument associated. Any restrictions on the passing method apply to the target of the actual argument.

F2003

Except in references to intrinsic inquiry functions, if the dummy argument is not allocatable and the actual argument is allocatable, the actual argument must be allocated.

If the dummy argument has the **VALUE** attribute it becomes associated with a definable anonymous data object whose initial value is that of the actual argument. Subsequent changes to the value or definition status of the dummy argument do not affect the actual argument.

F2003

If the dummy argument is neither a target nor a pointer, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.

If both the dummy and actual arguments are targets (with the **TARGET** attribute), in the following table, when all the conditions listed on the left apply, the associations listed on the right occur:

Conditions for dummy and actual arguments that are both targets	Associations
1. The dummy argument does not have the **VALUE** attribute. **1** 2. The actual argument is simply contiguous or the dummy argument is a scalar or an assumed-shape array that does not have the **CONTIGUOUS** attribute. **2** 3. The actual argument is not a coindexed object or an array section with a vector subscript.	1. Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure. 2. When execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

Conditions for dummy and actual arguments that are both targets	Associations
1. The dummy argument is an explicit-shape array, an assumed-shape array with the **CONTIGUOUS** attribute **2**, or an assumed-size array. 2. The actual argument is not simply contiguous. **2** 3. The actual argument is not an array section with a vector subscript.	1. Whether any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure is processor dependent. 2. When execution of the procedure completes, whether any pointers associated with the dummy argument remain associated with the actual argument is processor dependent.

Notes:
   1 Fortran 2003
   2 Fortran 2008	

If the dummy argument is a target and the corresponding actual argument is not a target or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

F2003 If the dummy argument has the **TARGET** attribute and the **VALUE** attribute, any pointers associated with the dummy argument become undefined when execution of the procedure completes. F2003

Allocatable objects as dummy arguments (Fortran 2003)

F2008 An actual argument that is associated with an allocatable dummy argument must be a variable. F2008 If the allocatable dummy argument is an array, the associated actual argument must also be an array of the same rank. The nondeferred type parameters of the actual argument must agree with those of the dummy argument.

An actual argument associated with a dummy argument that is allocatable must have deferred the same type parameters as the dummy argument.

On procedure entry, the allocation status of an allocatable dummy argument becomes that of the associated actual argument. If the dummy argument is **INTENT(OUT)** and the associated actual argument is allocated, the actual argument is deallocated on procedure invocation so that the dummy argument has an allocation status of not allocated. If the dummy argument is not **INTENT(OUT)** and the actual argument is allocated, the value of the dummy argument is that of the associated actual argument.

While the procedure is active, an allocatable dummy argument that does not have **INTENT(IN)** may be allocated, deallocated, defined, or become undefined. No reference to the associated actual argument is permitted via another alias if any of these events occur.

On exit from the routine, the actual argument has the allocation status of the allocatable dummy argument (there is no change, of course, if the allocatable dummy argument has **INTENT(IN)**). The usual rules apply for propagation of the value from the dummy argument to the actual argument.

Automatic deallocation of the allocatable dummy argument does not occur as a result of execution of a **RETURN** or **END** statement in the procedure of which it is a dummy argument.

Note: An allocatable dummy argument that has the **INTENT(IN)** attribute must not have its allocation status altered within the called procedure. The main difference between such a dummy argument and a normal dummy argument is that it might be deallocated on entry (and throughout execution of the procedure).

Examples

```
SUBROUTINE LOAD(ARRAY, FILE)
   REAL, ALLOCATABLE, INTENT(OUT) :: ARRAY(:, :, :)
   CHARACTER(LEN=*), INTENT(IN) :: FILE
   INTEGER UNIT, N1, N2, N3
   INTEGER, EXTERNAL :: GET_LUN
   UNIT = GET_LUN() ! Returns an unused unit number
   OPEN(UNIT, FILE=FILE, FORM='UNFORMATTED')
   READ(UNIT) N1, N2, N3
   ALLOCATE(ARRAY(N1, N2, N3))
   READ(UNIT) ARRAY
   CLOSE(UNIT)
END SUBROUTINE LOAD
```

Pointers as dummy arguments

The following requirements apply to actual arguments that correspond to dummy data pointers:

- If a dummy argument is a pointer, the actual argument must be a pointer `F2008` unless the dummy argument has the INTENT(IN) attribute and the actual argument has the TARGET attribute. `F2008` The type, nondeferred type parameters, and rank of a dummy argument must match those of the corresponding actual argument.
- `F2008` An actual argument associated with a dummy argument that is a pointer and has the CONTIGUOUS attribute must be simply contiguous. `F2008`
- An actual argument associated with a dummy argument that is a pointer must have deferred the same type parameters as the dummy argument.
- The actual argument reference is to the pointer itself, not to its target. When the procedure is invoked:
  - The dummy argument acquires the pointer association status of the actual argument.
  - If the actual argument is associated, the dummy argument is associated with the same target.

The association status can change during execution of the procedure. When the procedure finishes executing, the dummy argument's association status becomes undefined, if it is associated.

`IBM` The passing method must be by reference; that is, **%VAL** must not be specified for the pointer actual argument. `IBM`

Related information

- `F2008` Contiguity `F2008`

Procedures as dummy arguments

A dummy argument that is identified as a procedure `F2003` or a procedure pointer `F2003` is called a dummy procedure or `F2003` dummy procedure pointer, `F2003` respectively.

`F2003` If a dummy argument is a dummy procedure without the **POINTER** attribute, the associated actual argument must be the specific name of an external procedure, module procedure, dummy procedure, or intrinsic procedure whose name can be passed as an argument, an associated procedure pointer, or a reference to a function that returns an associated procedure pointer. If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

If a dummy argument is a procedure pointer, the associated actual argument must be a procedure pointer, a reference to a function that returns a procedure pointer, or a reference to the **NULL** intrinsic function. `F2003`

If an external procedure name or a dummy procedure name is used as an actual argument, its interface must be explicit or it must be explicitly declared with the **EXTERNAL** attribute.

If the interface of the dummy argument is explicit, the characteristics must be the same for the associated actual argument and the corresponding dummy argument, except that a pure actual argument may be associated with a dummy argument that is not pure.

If the interface of the dummy argument is implicit and either the name of the dummy argument is explicitly typed or it is referenced as a function, the dummy argument must not be referenced as a subroutine and the actual argument must be a function, `F2003` function procedure pointer `F2003`, or dummy procedure.

If the interface of the dummy argument is implicit and a reference to it appears as a subroutine reference, the actual argument must be a subroutine, `F2003` subroutine procedure pointer `F2003`, or dummy procedure.

Internal subprograms cannot be associated with a dummy procedure argument. You cannot use a non-intrinsic elemental procedure as an actual argument in Fortran 95.

Examples of procedures as dummy arguments

```
PROGRAM MYPROG
INTERFACE
  SUBROUTINE SUB (ARG1)
    EXTERNAL ARG1
    INTEGER ARG1
  END SUBROUTINE SUB
END INTERFACE
EXTERNAL IFUNC, RFUNC
REAL RFUNC

CALL SUB (IFUNC)    ! Valid reference
CALL SUB (RFUNC)    ! Invalid reference
!
! The first reference to SUB is valid because IFUNC becomes an
! implicitly declared integer, which then matches the explicit
! interface. The second reference is invalid because RFUNC is
! explicitly declared real, which does not match the explicit
! interface.
END PROGRAM
```

```
      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
```

Related information
- See Chapter 14, "Intrinsic procedures," on page 567 for details on which intrinsic procedures can be passed as actual arguments.
- See "Procedure references" on page 186 for the rules and restrictions for referencing a procedure.

Asterisks as dummy arguments

A dummy argument that is an asterisk can only appear 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, which indicates the statement label of a branch target statement in the same scope as the **CALL** statement, to which control is returned.

Examples
```
      CALL SUB(*10)
      STOP                  ! STOP is never executed
10    PRINT *, 'RETURN 1'
      CONTAINS
        SUBROUTINE SUB(*)
           ...
          RETURN 1          ! Control returns to statement with label 10
        END SUBROUTINE
      END
```

Resolution of procedure references

The subprogram name in a procedure reference is either established to be generic, established to be only specific, or not established.

A subprogram name is established to be generic in a scoping unit if one or more of the following cases is true:
- The scoping unit has an interface block with that name.
- The name of the subprogram is the same as the name of a generic intrinsic procedure that is specified in the scoping unit with the **INTRINSIC** attribute.
- The scoping unit accesses the generic name from a module through use association.
- There are no declarations of the subprogram name in the scoping unit, but the name is established to be generic in the host scoping unit.

A subprogram name is established to be only specific in a scoping unit when it has not been established to be generic and one of the following cases is true:
- An interface body in the scoping unit has the same name.

- There is a statement function, module procedure, or an internal subprogram in the scoping unit that has the same name.
- The name of the subprogram is the same as the name of a specific intrinsic procedure that is specified with the **INTRINSIC** attribute in the scoping unit.
- The scoping unit contains an **EXTERNAL** statement with the subprogram name.
- The scoping unit accesses the specific name from a module through use association.
- There are no declarations of the subprogram name in the scoping unit, but the name is established to be specific in the host scoping unit.

If a subprogram name is not established to be either generic nor specific, it is not established.

Rules for resolving procedure references to names

The following rules are used to resolve a procedure reference to a name established to be generic:

1. If there is an interface block with that name in the scoping unit or accessible through use association, and the reference is consistent with a non-elemental reference to one of the specific interfaces of that interface block, the reference is to the specific procedure associated with the specific interface.
2. If rule 1 does not apply, there is an interface block with that name in the scoping unit or accessible through use association, and the reference is consistent with an elemental reference to one of the specific interfaces of that interface block, the reference is to the specific elemental procedure associated with the specific interface.
3. If neither Rule 1 nor Rule 2 applies, the reference is to an intrinsic procedure if the procedure name in the scoping unit is specified with the **INTRINSIC** attribute or accesses a module entity whose name is specified with the **INTRINSIC** attribute, and the reference is consistent with the interface of that intrinsic procedure.
4. If Rule 1, Rule 2 and Rule 3 do not apply, but the name is established to be generic in the host scoping unit, the name is resolved by applying the rules to the host scoping unit. For this rule to apply, there must be agreement between the host scoping unit and the scoping unit of which the name is either a function or a subroutine.

The following rules are used to resolve a procedure reference to a name established to be only specific:

1. If the scoping unit is a subprogram, and it contains either an interface body with that name or the name has the **EXTERNAL** attribute, and if the name is a dummy argument of that subprogram, the dummy argument is a dummy procedure. The reference is to that dummy procedure.
2. If Rule 1 does not apply, and the scoping unit contains either an interface body with that name or the name has the **EXTERNAL** attribute, the reference is to an external subprogram.
3. In the scoping unit, if a statement function or internal subprogram has that name, the reference is to that procedure.
4. In the scoping unit, if the name has the **INTRINSIC** attribute, the reference is to the intrinsic procedure with that name.

5. The scoping unit contains a reference to a name that is the name of a module procedure that is accessed through use association. Because of possible renaming in the **USE** statement, the name of the reference may differ from the original procedure name.

6. If none of these rules apply, the reference is resolved by applying these rules to the host scoping unit.

The following rules are used to resolve a procedure reference to a name that is not established:

1. If the scoping unit is a subprogram and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure. The reference is to that dummy procedure.

2. If Rule 1 does not apply, and the name is the name of an intrinsic procedure, the reference is to that intrinsic procedure. For this rule to apply, there must be agreement between the intrinsic procedure definition and the reference that the name is either a function or subroutine.

3. If neither Rule 1 nor 2 applies, the reference is to the external procedure with that name.

Recursion

A procedure that can reference itself, directly or indirectly, is called a recursive procedure. Such a procedure can reference itself indefinitely until a specific condition is met. For example, you can determine the factorial of the positive integer N as follows:

```
INTEGER N, RESULT
READ (5,*) N
IF (N.GE.0) THEN
  RESULT = FACTORIAL(N)
END IF
CONTAINS
  RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
    INTEGER RES
    IF (N.EQ.0) THEN
      RES = 1
    ELSE
      RES = N * FACTORIAL(N-1)
    END IF
  END FUNCTION FACTORIAL
END
```

For details on syntax and rules, see "FUNCTION" on page 379, "SUBROUTINE" on page 468, or "ENTRY" on page 358.

IBM You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the procedure specifies either the **RECURSIVE** or **RESULT** keyword. **IBM**

Pure procedures

Pure procedures are free of side effects and are particularly useful in **FORALL** statements and constructs, which by design require that all referenced procedures be free of side effects.

A pure procedure is a procedure that satisfies one of the following conditions:

• The procedure has the **PURE** prefix specifier.

- The procedure has the **ELEMENTAL** prefix specifier `F2008` and does not have the **IMPURE** specifier `F2008` .

A procedure must be pure in the following contexts:
- An internal procedure of a pure procedure
- A procedure referenced in the *scalar_mask_expr* or body of a **FORALL** statement or construct, including one referenced by a defined operator, defined assignment, or finalization
- A procedure referenced in a pure procedure
- A procedure actual argument to a pure procedure

Intrinsic functions (except **RAND**, an XL Fortran extension) and the **MOVE_ALLOC** and **MVBITS** subroutines are always pure. They do not need to be explicitly declared to be pure. A statement function is pure if and only if all functions that it references are pure.

The *specification_part* of a pure function must specify that all dummy arguments have an **INTENT(IN)**, except procedure arguments, arguments with the **POINTER** attribute, `F2008` and arguments with the **VALUE** attribute `F2008` . The *specification_part* of a pure subroutine must specify the intents of all dummy arguments, except for procedure arguments, asterisks, arguments that have the **POINTER** attribute, `F2008` and arguments that have the **VALUE** attribute `F2008` . Any interface body for such pure procedures must similarly specify the intents of its dummy arguments.

The *execution_part* and *internal_subprogram_part* of a pure procedure cannot refer to a dummy argument with an **INTENT(IN)**, a global variable (or any object that is storage associated with one), or any subobject thereof, in contexts that may cause its value to change: that is, in contexts that produce side effects. The *execution_part* and *internal_subprogram_part* of a pure function must not use a dummy argument, a global variable, or an object that is associated with a global variable, or a subobject thereof, in the following contexts:
- As *variable* in an assignment statement, or as *expression* in an assignment statement if *variable* is of a derived type that has a pointer component at any level
- As *pointer_object* or *target* in a pointer assignment statement
- As a **DO** or implied-**DO** variable
- As an *input_item* in a **READ** statement
- As an internal file identifier in a **WRITE** statement
- As an **IOSTAT=**, **SIZE=** or **IOMSG=** specifier variable in an input/output statement
- As a variable in an **ALLOCATE**, **DEALLOCATE**, **NULLIFY**, or **ASSIGN** statement
- As an actual argument that is associated with a dummy argument with the **POINTER** attribute or with an intent of **OUT** or **INOUT**
- As the argument to **LOC**
- As a **STAT=** or **ERRMSG=** specifier
- As a variable in a **NAMELIST** which appears in a **READ** statement
- A variable that is the selector in a **SELECT TYPE** or **ASSOCIATE** construct if the associate name of that construct appears in a variable definition context.

A pure procedure must not specify that any entity is **VOLATILE**. In addition, it must not contain any references to data that is **VOLATILE**, that would otherwise be accessible through use- or host-association. This includes references to data which occur through **NAMELIST I/O**.

F2008 A local variable of a pure subprogram or a local variable of a **BLOCK** construct within a pure subprogram cannot have the **SAVE** attribute. F2008

Only internal I/O is permitted in pure procedures. Therefore, the unit identifier of an I/O statement cannot be an asterisk (*) or refer to an external unit. The I/O statements are as follows:

- **BACKSPACE**
- **CLOSE**
- **ENDFILE**
- F2003 **FLUSH** F2003
- **INQUIRE**
- **OPEN**
- **PRINT**
- **READ**
- **REWIND**
- **WAIT**
- **WRITE**

The **PAUSE** and **STOP** statements are not permitted in pure procedures.

There are two differences between pure functions and pure subroutines:

1. Subroutine nonpointer dummy data objects may have any intent, while function nonpointer dummy data objects must be **INTENT(IN)**.
2. Subroutine dummy data objects with the **POINTER** attribute can change association status and/or definition status

If a procedure is not defined as pure, it must not be declared pure in an interface body. However, the converse is not true: if a procedure is defined as pure, it does not need to be declared pure in an interface body. Of course, if an interface body does not declare that a procedure is pure, that procedure (when referenced through that explicit interface) cannot be used as a reference where only pure procedure references are permitted (for example, in a **FORALL** statement).

Examples

```
PROGRAM ADD
  INTEGER ARRAY(20,256)
  INTERFACE                             ! Interface required for
    PURE FUNCTION PLUS_X(ARRAY)         !   a pure procedure
      INTEGER, INTENT(IN) :: ARRAY(:)
      INTEGER :: PLUS_X(SIZE(ARRAY))
    END FUNCTION
  END INTERFACE
  INTEGER :: X
  X = ABS(-4)                           ! Intrinsic function
                                        !   is always pure

  FORALL (I=1:20, I /= 10)
    ARRAY(I,:) = I + PLUS_X(ARRAY(I,:)) ! Procedure references in
                                        !   FORALL must be pure

  END FORALL
END PROGRAM
PURE FUNCTION PLUS_X(ARRAY)
```

```
      INTEGER, INTENT(IN) :: ARRAY(:)
      INTEGER :: PLUS_X(SIZE(ARRAY)),X
      INTERFACE
        PURE SUBROUTINE PLUS_Y(ARRAY)
          INTEGER, INTENT(INOUT) :: ARRAY(:)
        END SUBROUTINE
      END INTERFACE
      X=8
      PLUS_X = ARRAY+X
      CALL PLUS_Y(PLUS_X)
    END FUNCTION

    PURE SUBROUTINE PLUS_Y(ARRAY)
      INTEGER, INTENT(INOUT) :: ARRAY(:)      ! Intent must be specified
      INTEGER :: Y
      Y=6
      ARRAY = ARRAY+Y
    END SUBROUTINE
```

Elemental procedures

An elemental procedure is an elemental intrinsic procedure or an elemental
subprogram that has the **ELEMENTAL** prefix specifier. If the **ELEMENTAL** prefix
specifier is used, the **RECURSIVE** specifier cannot be used.

You cannot use the **-qrecur** option when you specify elemental procedures.

An elemental subprogram is a pure subprogram unless it has the `F2008` **IMPURE**
`F2008` prefix specifier.

Elemental procedures must conform to the following rules:
- The result of an elemental function must be a scalar, and must not have the
  `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute.
- The following apply to dummy arguments used in elemental subprograms:
  - All dummy arguments must be scalar data objects and must not have the
    `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute.
  - The result of an elemental function must be scalar and must not have the
    `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute.
  - If a dummy argument is the base object of an object designator that appears
    in the specification expression that specifies a type parameter value of the
    result of the elemental function, the object designator must appear only as the
    subject of a specification inquiry, and the inquiry must not depend on a
    property that is deferred.
  - A dummy argument cannot be an asterisk.
  - `F2008` A dummy argument that does not have the **VALUE** attribute must
    have the **INTENT** attribute specified. `F2008`
- Elemental subprograms can have **ENTRY** statements, but the **ENTRY** statement
  cannot have the **ELEMENTAL** prefix. The procedure defined by the **ENTRY**
  statement is elemental if the **ELEMENTAL** prefix is specified in the
  SUBROUTINE or **FUNCTION** statement.
- Elemental procedures can be used as defined operators in elemental expressions,
  but they must follow the rules for elemental expressions as described in
  "Operators and expressions" on page 105.

In a reference to an elemental procedure, the following rules apply:

- If any actual argument is an array, every actual argument that corresponds to an **INTENT(OUT)** or **INTENT(INOUT)** dummy argument must be an array.
- All actual arguments must be conformable.

In a reference to an elemental function, the following rules apply:
- The shape of the result is the same as the shape of the actual argument with the greatest rank.
- If there are no actual arguments or all the actual arguments are scalars, the result is scalar.

If any actual argument of an elemental procedure is an array, the value of the elements, if any, of the results are the same as would be obtained if the procedure had been applied separately, in the element order, to the corresponding elements of each array actual argument.

Examples

Example 1:

```
! Example of an elemental function
PROGRAM P
INTERFACE
  ELEMENTAL REAL FUNCTION LOGN(X,N)
     REAL, INTENT(IN) :: X
     INTEGER, INTENT(IN) :: N
  END FUNCTION LOGN
END INTERFACE

REAL RES(100), VAL(100,100)
  ...
DO I=1,100
   RES(I) = MAXVAL( LOGN(VAL(I,:),2) )
END DO
    ...
END PROGRAM P
```

Example 2:

```
! Elemental procedure declared with a generic interface
INTERFACE RAND
   ELEMENTAL FUNCTION SCALAR_RAND(x)
     REAL, INTENT(IN) :: X
   END FUNCTION SCALAR_RAND

   FUNCTION VECTOR_RANDOM(x)
     REAL X(:)
     REAL VECTOR_RANDOM(SIZE(x))
   END FUNCTION VECTOR_RANDOM
END INTERFACE RAND

REAL A(10,10), AA(10,10)

! The actual argument AA is a two-dimensional array. The procedure
! taking AA as an argument is not declared in the interface block.
! The specific procedure SCALAR_RAND is then called.

A = RAND(AA)


! The actual argument is a one-dimensional array section. The procedure
! taking a one-dimensional array as an argument is declared in the
! interface block. The specific procedure VECTOR_RANDOM is then called.
```

```
! This is a non-elemental reference since VECTOR_RANDOM is not elemental.

A(:,1) = RAND(AA(6:10,2))
END
```

Related information

- "ALLOCATABLE (Fortran 2003)" on page 285
- "BIT_SIZE(I)" on page 591
- "ENTRY" on page 358
- "FUNCTION" on page 379
- "INTENT" on page 403
- Chapter 14, "Intrinsic procedures," on page 567
- "KIND(X)" on page 641
- "LEN(STRING, KIND)" on page 643
- "MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)" on page 670
- "POINTER (Fortran 90)" on page 426
- "Pure procedures" on page 208
- "SUBROUTINE" on page 468
- "VALUE (Fortran 2003)" on page 487

Chapter 9. XL Fortran Input/Output

XL Fortran supports both synchronous and asynchronous input/output (I/O). Synchronous I/O halts an executing application until I/O operations complete. Asynchronous I/O allows an application to continue processing while I/O operations occur in the background. Both I/O types support the following file access methods:

- Sequential access
- Direct access
- ▶F2003◀ Stream access ▶F2003◀

Each method of access offers benefits and limitations based on the I/O concepts of, Records, Files and Units.

This section also provides explanations of the **IOSTAT=** specifier codes that can result when using XL Fortran I/O statements.

Records

A record contains a sequence of characters or values. XL Fortran supports three record types:

- formatted
- unformatted
- endfile

Formatted records

A formatted record consists of a sequence of ASCII characters that can print in a readable format. Reading a formatted record converts the data values from readable characters into an internal representation. Writing a formatted record converts the data from the internal representation into characters.

Unformatted records

An unformatted record contains a sequence of values in an internal representation that can contain both character and noncharacter data. An unformatted record can also contain no data. Reading or writing an unformatted record does not convert any data the record contains from the internal representation.

Endfile records

If it exists, an endfile record is the last record of a file. It has no length. It can be written explicitly by an **ENDFILE** statement. It can be written implicitly to a file connected for sequential access when the last data transfer statement was a **WRITE** statement, no intervening file positioning statement referring to the file has been executed, and the following is true:

- A **REWIND** or **BACKSPACE** statement references the unit to which the file is connected; or
- The file is closed, either explicitly by a **CLOSE** statement, implicitly by a program termination not caused by an error condition, or implicitly by another **OPEN** statement for the same unit.

Files

A file is an internal or external sequence of records or file storage units. You determine the file access method when connecting a file to a unit. You can access an external file using three methods:

- Sequential access
- Direct access
- F2003 Stream access F2003

You can only access an internal file sequentially.

Definition of an external file

You must associate an external file with an I/O device such as a disk, or terminal. An external file exists for a program when a program creates that file, or the file is available to that program for reading and writing. Deleting an external file ends the existence of that file. An external file can exist and contain no records.

IBM To specify an external file by a file name, you must designate a valid operating system file name. Each file name can contain a maximum of 255 characters. If you specify a full path name, it can contain a maximum of 4095 characters. IBM

The preceding I/O statement determines the position of an external file. You can position an external file to:

- The initial point, which is the position immediately before the first record, or the first file storage unit.
- The terminal point, which is the position immediately after the last record, or the last file storage unit.
- The current record, when the file position is within a record. Otherwise, there is no current record.
- The preceding record, which is the record immediately before the current record. If there is no current record, the preceding record is the record immediately before the current file position. A preceding record does not exist when the file position is at its initial point or within the first record of the file.
- The next record, which is the record immediately after the current record. If there is no current record, the next record is the record immediately after the current position. The next record does not exist when the file position is at the terminal point or within the last record of the file.

An external file can also have indeterminate position after an error.

File access methods

Sequential access

Using sequential access, records in a file are read or written based on the logical order of records in that file. Sequential access supports both internal and external files.

External files: 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 F2003 or stream access F2003 I/O statements during the time the file is connected for sequential access.

Internal files: An internal file is a character variable that is not an array section with a vector subscript. You do not need to create internal files. They always exist, and are available to the application.

If an internal file is a scalar character variable, the file consists of one record with a length equal to that of the scalar variable. 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.

An internal file must contain only formatted records. **READ** and **WRITE** are the only statements that can specify an internal file. If a **WRITE** statement writes less than an entire record, blanks fill the remainder of that record.

F2003 An internal file is positioned at the beginning of the first record prior to data transfer, except for child data transfer statements. This record becomes the current record. F2003

Direct access

Using direct access, the records of an external file can be read or written in any order. The records must be either all formatted or all unformatted. The records must not be read or written using sequential or stream access, list-directed or namelist formatting, or a nonadvancing input/output statement. If the file was previously connected for sequential access, the last record of the file is an endfile record. The endfile record is not considered a part of the file 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 when the file is connected.

Records in a file connected for direct access cannot be deleted, but they can be rewritten with a new value. A record cannot be read unless it has first been written.

Stream access (Fortran 2003)

You can connect external files for stream access as either formatted or unformatted. Both forms use external stream files composed of one byte file storage units. While a file connected for unformatted stream access has only a stream structure, files connected for formatted stream access have both a record and a stream structure. These dual structure files have the following characteristics:

* Some file storage units represent record markers.
* The record structure is inferred from the record markers stored in the file.
* There is no theoretical limit on record length.
* Writing an empty record without a record marker has no effect.
* If there is no record marker at the end of a file, the final record is incomplete but not empty.
* The endfile record in a file previously connected for sequential access is not considered part of the file when you connect that file for stream access.

The first file storage unit of a file connected for formatted stream access has a position of 1. The position of each subsequent storage unit is greater than the storage unit immediately before it. The positions of successive storage units are not always consecutive and positionable files need not be read or written to in order of position. To determine the position of a file storage unit connected for formatted stream access, use the **POS=** specifier of the **INQUIRE** statement. If the file can be positioned, you can use the value obtained using the **INQUIRE** statement to position that file. You read from the file while connected to the file, as long as the storage unit has been written to since file creation and that the connection permits a **READ** statement. File storage units of a file connected for formatted stream access can only be read or written by formatted stream access input/output statements.

The first file storage unit of a file connected for unformatted stream access has a position of 1. The position value of successive storage units is incrementally one greater than the storage unit it follows. Positionable files need not be read or written to in order of position. Any storage unit can be read from the file while connected to the file, if the storage unit has been written to since file creation and that the connection permits a **READ** statement. File storage units of a file connected for unformatted stream access can only be read or written by stream access input/output statements.

Units

A unit is a means of referring to a file. Programs refer to files by the unit numbers indicated by unit specifiers in input/output statements. See **[UNIT=]** for the form of a unit specifier.

Connection of a unit

A connection refers to the association between a file and a unit. A connection must occur before the records of a file can be read or written.

There are three ways to connect a file to a unit:
- Preconnection
- **IBM** Implicit connection **IBM**
- Explicit connection, using the **OPEN** statement

Preconnection
Preconnection occurs when the program begins executing. You can specify preconnection in I/O statements without the prior execution of an **OPEN** statement. Three units are preconnected for formatted sequential access to the standard error, input, and output devices. **F2003** You can refer to these units in input/output statements using the ERROR_UNIT, INPUT_UNIT, and OUTPUT_UNIT constants from the ISO_FORTRAN_ENV module. **F2003**

IBM extension

You can also refer to these units directly using the following values:
- Unit *0* for the standard error device
- Unit *5* for the standard input device
- Unit *6* for the standard output device

The preconnected units use the default specifier values for the **OPEN** statement with the following exceptions:

- **STATUS='OLD'**
- **ACTION='READWRITE'**
- **FORM='FORMATTED'**

---- **End of IBM extension** ----

Implicit connection (IBM extension)

Implicit connection occurs when a sequential statement that is; **ENDFILE**, **PRINT**, **READ**, **REWIND**, or **WRITE** executes on a unit not already connected to an external file. The executing statement connects that unit to a file with a predetermined name. By default, this connection is unit *n* to file *fort.n*. You do not need to create the file before implicit connection. To implicitly connect to a different file name, refer to the **UNIT_VARS** run-time option under *Setting runtime options* in the *XL Fortran Compiler Reference*.

You can not specify unit *0* for implicit connection.

You can only connect a preconnected unit implicitly if you terminate the connection between the unit and the external file. In the next example a preconnected unit closes before implicit connection takes place.

Sample Implicit Connection

```
      PROGRAM TRYME
      WRITE ( 6, 10 ) "Hello1"   ! "Hello1" written to standard output
      CLOSE ( 6 )
      WRITE ( 6, 10 ) "Hello2"   ! "Hello2" written to fort.6
10    FORMAT (A)
      END
```

A unit with an implicit connection uses the default specifier values of the **OPEN** statement, except for the **FORM=** and **ASYNCH=** specifiers. The first data transfer statement determines the values for **FORM=** and **ASYNCH=**.

If the first I/O statement uses format-directed, list-directed, or namelist formatting, the value of the **FORM=** specifier is set to **FORMATTED**. An unformatted I/O statement sets the specifier to **UNFORMATTED**.

If the first I/O statement is asynchronous, the value of the **ASYNCH=** specifier is set to **YES**. A synchronous I/O statement sets the specifier to **NO**.

Disconnection

The **CLOSE** statement disconnects a file from a unit. You can connect the file again within the same program to the same unit or to a different unit. You can connect the unit again within the same program to the same file or a different file.

IBM
- You can not close unit *0*
- You can not reconnect unit *5* to standard input after the unit closes
- You can not reconnect unit *6* to standard output after the unit closes **IBM**

Data transfer statements

The **READ** statement obtains data from an external or internal file and transfers the data to internal storage. If you specify an input list, values transfer from the file to the data items you specify.

The **WRITE** statement transfers data from internal storage into an external or internal file.

The **PRINT** statement transfers data from internal storage into an external file. Specifying the **–qport=typestmt** compiler option enables the **TYPE** statement which supports functionality identical to **PRINT**. If you specify an output list and format specification, values transfer to the file from the data items you specify. If you do not specify an output list, the **PRINT** statement transfers a blank record to the output device unless the **FORMAT** statement it refers to contains, as the first specification, a character string edit descriptor or a slash edit descriptor. In this case, the records these specifications indicate transfer to the output device.

Execution of a **WRITE** or **PRINT** statement for a file that does not exist creates that file, unless an error occurs.

If an input/output item is a pointer, data is transferred between the file and the associated target.

F2003 If an input or output item is polymorphic, or is a derived type with a pointer or an allocatable component, it must be processed by a user-defined derived-type input/output procedure. F2003

During advancing input from a file with a **PAD=** specifier that has the value **NO**, the input list and format specification must not require more characters from the record than that record contains. If the **PAD=** specifier has the value **YES**, blank characters are supplied if the input list and format specification require more characters from the record than the record contains.

IBM If you want to pad files connected for sequential access, specify the **-qxlf77=noblankpad** compiler option. This compiler option also sets the default value for the **PAD=** specifier to **NO** for direct and stream files and **YES** for sequential files. IBM

During nonadvancing input from a file with a **PAD=** specifier that has the value **NO**, an end-of-record condition occurs if the input list and format specification require more characters from the record than the record contains. If the **PAD=** specifier has the value **YES**, an end-of-record condition occurs and blank characters are supplied if an input item and its corresponding data edit descriptor require more characters from the record than the record contains. If the record is the last record of a stream file, an end-of-file condition occurs.

Asynchronous Input/Output

You can specify asynchronous **READ** and **WRITE** data transfer statements to initiate asynchronous data transfer. Execution continues after the asynchronous I/O statement, without waiting for the data transfer to complete.

Executing a matching **WAIT** statement with the same **ID=** value that was returned to the **ID=** variable in the data transfer statement detects that the data transfer statement is complete, or waits for that data transfer statement to complete.

The data transfer of an I/O item in an asynchronous I/O statement can complete:
- During the execution of the asynchronous data transfer statement
- At any time before the execution of the matching **WAIT** statement
- During the matching **WAIT** statement

For information on situations where data transfer must complete during the asynchronous data transfer statement, see *Implementation details of XL Fortran Input/Output* in the *XL Fortran Optimization and Programming Guide*.

If an error occurs during the execution of an asynchronous data transfer statement, the variable associated with the **ID=** specifier remains undefined. The **IOSTAT=** specifier indicates the status of the I/O operation and control is transferred to the statement specified by the **ERR=** specifier.

You must not reference, define, or undefine variables or items associated with a variable appearing in an I/O list for an asynchronous data transfer statement, until the execution of the matching **WAIT** statement.

Any deallocation of allocatable objects and pointers and changing association status of pointers are disallowed between an asynchronous data transfer statement and the matching **WAIT** statement.

F2003 Multiple outstanding data transfer operations on the same unit can be both **READ** and **WRITE**. A **WAIT** statement will perform a wait operation for all pending data transfers for the specified unit if the **ID=** specifier is omitted. F2003

In the case of direct access, an asynchronous **WRITE** statement must not specify both the same unit and record number as any asynchronous **WRITE** statement for which the matching **WAIT** statement has not been executed. F2003 For stream access, an asynchronous **WRITE** statement must not specify either the same unit and location within a file as any asynchronous **WRITE** statement for which the matching **WAIT** statement has not been executed. F2003

In the portion of the program that executes between the asynchronous data transfer statement and the matching **WAIT** statement, you must not reference, define, or undefine variables or items associated with the *integer_variable* in the **NUM=** specifier of that data transfer statement.

Using Asynchronous I/O

```
SUBROUTINE COMPARE(ISTART, IEND, ISIZE, A)
INTEGER, DIMENSION(ISIZE) :: A
INTEGER I, ISTART, IEND, ISIZE
DO I = ISTART, IEND
  IF (A (I) /= I) THEN
    PRINT *, "Expected ", I, ", got ", A(I)
  END IF
END DO
END SUBROUTINE COMPARE

PROGRAM SAMPLE
INTEGER, PARAMETER :: ISIZE = 1000000
INTEGER, PARAMETER :: SECT1 = (ISIZE/2) - 1, SECT2 = ISIZE - 1
INTEGER, DIMENSION(ISIZE), STATIC :: A
INTEGER IDVAR

OPEN(10, STATUS="OLD", ACCESS="DIRECT", ASYNCH="YES", RECL=(ISIZE/2)*4)
A = 0

! Reads in the first part of the array.

READ(10, REC=1) A(1:SECT1)

! Starts asynchronous read of the second part of the array.

READ(10,ID=IDVAR, REC=2) A(SECT1+1:SECT2)
```

```
! While the second asynchronous read is being performed,
! do some processing here.

CALL COMPARE(1, SECT1, ISIZE, A)

WAIT(ID=IDVAR)

CALL COMPARE(SECT1+1, SECT2, ISIZE, A)
END
```

Advancing and nonadvancing Input/Output

Advancing I/O positions the file after the last record that is read or written, unless an error condition occurs.

Nonadvancing I/O can position the file at a character position within the current record, or a subsequent record. With nonadvancing I/O, you can **READ** or **WRITE** a record of the file by a sequence of I/O statements that each access a portion of the record. You can also read variable-length records and about the length of the records.

Nonadvancing I/O

```
! Reads digits using nonadvancing input

    INTEGER COUNT
    CHARACTER(1) DIGIT
    OPEN (7)
    DO
     READ (7,FMT="(A1)",ADVANCE="NO",EOR=100) DIGIT
       COUNT = COUNT + 1
     IF ((ICHAR(DIGIT).LT.ICHAR('0')).OR.(ICHAR(DIGIT).GT.ICHAR('9'))) THEN
       PRINT *,"Invalid character ", DIGIT, " at record position ",COUNT
       STOP
     END IF
    END DO
100 PRINT *,"Number of digits in record = ", COUNT
    END

! When the contents of fort.7 is '1234\n', the output is:

!   Number of digits in record =  4
```

User-defined derived-type Input/Output procedure interfaces (Fortran 2003)

User-defined derived-type input/output procedures allow a program to override the default handling of derived-type objects and values in data transfer input/output statements.

A user-defined derived-type input/output procedure is a procedure accessible by a *dtio_generic_spec*. A particular user-defined derived-type input/output procedure is selected based on the existence of one of the following:

1. A suitable generic interface with both:
    a. a *dtio_generic_spec* that is appropriate to the direction (read or write) and form (formatted or unformatted) of the data transfer, and
    b. a specific interface whose *dtv* argument is compatible with the effective item. For more information on *dtv* see "User-defined derived-type Input/Output procedures (Fortran 2003)" on page 173.
2. A suitable generic binding for the declared type of the effective item.

If a derived-type input/output procedure is selected as specified above, it is called for any appropriate data transfer input/output statements executed in that scoping unit. The procedure controls the actual data transfer operations for the derived-type input/output list item.

A data transfer statement that includes a derived-type input/output list item and that causes a user-defined derived-type input/output procedure to be invoked is called a parent data transfer statement. A data transfer statement that is executed while a parent data transfer statement is being processed, and that specifies the unit passed into a user-defined derived-type input/output procedure, is called a child data transfer statement.

A child data transfer statement is processed differently from a nonchild data transfer statement in the following ways:
- Executing a child data transfer statement does not position the file prior to data transfer.
- An unformatted child data transfer statement does not position the file after data transfer is complete.

User-defined derived-type Input/Output (Fortran 2003)

For a particular derived type and a particular set of kind type parameter values, there are four possible user-defined derived-type input/output procedures: one each for formatted input, formatted output, unformatted input, and unformatted output. You do not need to supply all four procedures. You can specify the procedures to be used for derived-type input/output by interface blocks or by generic bindings, with a *dtio_generic_spec* (the values for *dtio_generic_spec* are given in Table 21 on page 173).

While a parent data transfer statement is active, the following rules apply:
- When a parent **READ** statement is active, an input/output statement does not read from any external unit other than the one specified by the dummy argument *unit* and does not write to any external unit.
- When a parent **WRITE** or **PRINT** statement is active, an input/output statement does not write to any external unit other than the one specified by the dummy argument *unit* and does not read from any external unit.
- A data transfer statement that specifies an internal file is permitted.
- **OPEN**, **CLOSE**, **BACKSPACE**, **ENDFILE**, and **REWIND** statements are not executed.
- The user-defined procedure, and any procedures that it invokes, cannot define or undefine any storage location referenced by any input/output list item, the corresponding format, or any specifier in any active parent data transfer statement, except through the *dtv* argument.

The following are additional rules for user-defined derived-type input/output procedure data transfer statements:
- The procedure may use a **FORMAT** with a **DT** edit descriptor for handling a component of the derived type that is itself of a derived type. A child data transfer statement that is a list-directed or namelist input/output statement may contain a list item of derived type.
- Because a child data transfer statement does not position the file prior to data transfer, it starts transferring data from where the file was positioned by the

parent data transfer statement's most recently processed effective list item or record positioning edit descriptor. This is not necessarily at the beginning of a record.

- A record positioning edit descriptor, such as **TL** and **TR**, used on *unit* by a child data transfer statement, does not cause the record to be positioned before its position at the time the procedure was invoked.
- Parent and child data transfer statements cannot be asynchronous.
- A child data transfer statement must not specify the **ID=**, **POS=**, or **REC=** specifiers in an input/output control list.

Examples

```
! The example program reads "dt.input" and prints the content out.
MODULE m
  TYPE dt
    INTEGER i(4)
    CONTAINS
      PRIVATE
      PROCEDURE, PASS :: WRITE => writedt
      PROCEDURE, PASS :: READ => readdt
      GENERIC, PUBLIC :: WRITE(FORMATTED) => WRITE
      GENERIC, PUBLIC :: READ(FORMATTED)  => READ
  END TYPE

  CONTAINS
  SUBROUTINE writedt (dtv, unit, iotype, v_list, iostat, iomsg)
    CLASS(dt), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER(*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN)  :: v_list(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER(*), INTENT(INOUT) :: iomsg

    WRITE (unit, *, IOSTAT=iostat) dtv%I
  END SUBROUTINE

  SUBROUTINE readdt (dtv, unit, iotype, v_list, iostat, iomsg)
    CLASS(dt), INTENT(INOUT) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER(*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN)  :: v_list(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER(*), INTENT(INOUT) :: iomsg

    READ (unit, *, IOSTAT=iostat) dtv%I
  END SUBROUTINE
END MODULE

PROGRAM main
  USE m
  TYPE(dt) :: dt1

  OPEN (1, FILE='dt.input', FORM='FORMATTED')
  READ (1, FMT='(dt)', ADVANCE='NO') dt1
  CLOSE(1)
  WRITE(6, '(dt)') dt1
END PROGRAM main
```

File position before and after data transfer

For an explicit connection using an **OPEN** statement for sequential or stream I/O that specifies the **POSITION=** specifier, you can position the file explicitly at the beginning, at the end, where the position is on opening.

If the **OPEN** statement does not specify the **POSITION=** specifier:

- If the **STATUS=** specifier has the value **NEW** or **SCRATCH**, the file position is at the beginning.

IBM extension

- If you specify **STATUS='OLD'** with the **-qposition=appendold** compiler option, and the next operation that changes the file position is a **WRITE** statement, then the file position is at the end. If these conditions are not met, the file position is at the beginning.
- If you specify **STATUS='UNKNOWN'** with the **-qposition=appendunknown** compiler option, and the next operation is a **WRITE** statement, then the file position is at the end. If these conditions are not met, the file position is at the beginning.

After an implicit **OPEN**, the file position is at the beginning:

- If the first I/O operation on the file is **READ**, the application reads the first record of the file.
- If the first I/O operation on the file is **WRITE** or **PRINT**, the application deletes the contents of the file and writes at the first record.

End of IBM extension

You can use a **REWIND** statement to position a file at the beginning. The preconnected units 0, 5 and 6 are positioned as they come from the parent process of the application.

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

- Sequential access for an external file:
  - For advancing input, the file position is at the beginning of the next record. This record becomes the current record.
  - Advancing output creates a new record and becomes the last record of the file.
- Sequential access for an internal file:
  - File position is at the beginning of the first record of the file. This record becomes the current record.
- Direct access:
  - File position is at the beginning of the record that the **REC=** specifier indicates. This record becomes the current record.
- F2003 Stream access:
  - File position is immediately before the file storage unit the **POS=** specifier indicates. If there is no **POS=** specifier, the file position remains unchanged.

  F2003

F2003 File positioning for a child data transfer statement is processed differently from a nonchild data transfer statement in the following ways:

- Executing a child data transfer statement does not position the file prior to data transfer.
- An unformatted child data transfer statement does not position the file after data transfer is complete. F2003

After advancing I/O data transfer, the file position is:

- 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 or formatted stream access becomes the last record of the file.

After nonadvancing input the file position:
- If no error condition or end-of-file condition occurs, but an end-of-record condition occurs, the file position is immediately after the record read.
- If no error condition, end-of-file condition or end-of-record condition occurs in a nonadvancing input statement, the file position does not change.
- If no error condition occurs in a nonadvancing output statement, the file position does not change.
- In all other cases, the file position is immediately after the record read or written and that record becomes the preceding record.

If the file position is beyond the endfile record, a **READ**, **WRITE**, **PRINT**, or **ENDFILE** statement can not execute if the compiler option **-qxlf77=softeof** is not set. A **BACKSPACE** or **REWIND** statement can be used to reposition the file.

IBM Use the **-qxlf77=softeof** option to be able to read and write past the end-of-file. **IBM**

F2003 For formatted stream output with no errors, the terminal point of the file is set to the highest-numbered position to which data was transferred by the statement. For unformatted stream output with no errors, the file position is unchanged. If the file position exceeds the previous terminal point of the file, the terminal point is set to the file position. Use the **POS=** specifier with an empty output list to extend the terminal point of the file without writing data. After data transfer, if an error occurs, the file position is indeterminate. **F2003**

Conditions and IOSTAT values

An IOSTAT value is a value assigned to the variable for the **IOSTAT=** specifier if end-of-file condition, end-of-record condition or an error condition occurs during an input/output statement. The **IOSTAT=** specifier reports the following types of error conditions. If the input or output statement is successful, the **IOSTAT** value is 0.
- Catastrophic
- Severe
- Recoverable
- Conversion
- Language

End-of-record conditions

When an application encounters an end-of-record condition with the **IOSTAT=** specifier, it sets the value of the variable specified by the **IOSTAT=** specifier to -4 and branches to the **EOR=** label if that label is present. If the **IOSTAT=** and **EOR=** specifiers are not present on the I/O statement when an application encounters an end-of-record condition, the application stops.

Table 23. IOSTAT values for end-of-record conditions

IOSTAT Value	End-of-Record Condition Description
-4	End of record encountered on a nonadvancing, format-directed READ of an internal or external file.

End-of-file conditions

An end-of-file condition can occur in the following instances:

- At the beginning of the execution of an input statement.
- During execution of a formatted input statement that requires more than one record through the interaction of the input list and the format.
- During execution of a stream input statement.
- When encountering an endfile record while reading of a file connected for sequential access.
- When attempting to read a record beyond the end of an internal file.

F2003 For stream access, an end-of-file condition occurs when you attempt to read beyond the end of a file. An end-of-file condition also occurs if you attempt to read beyond the last record of a stream file connected for formatted access. F2003

An end-of-file condition causes **IOSTAT=** to be set to one of the values defined below and branches to the **END=** label if these specifiers are present on the input statement. If the **IOSTAT=** and **END=** specifiers are not present on the input statement when an end-of-file condition is encountered, the program stops.

Table 24. IOSTAT values for end-of-file conditions

IOSTAT Value	End-of-File Condition Description
-1	End of file encountered on sequential or stream READ of an external file, or END= is specified on a direct access read and the record is nonexistent.
-1 **1**	End of file encountered on READ of an internal file.
-2	End of file encountered on READ of an internal file.
Note:	
1 Fortran 2003. See the **IOSTAT_END** run-time option for more information.	

Error conditions

Catastrophic errors

Catastrophic errors are system-level errors encountered within the run-time system that prevent further execution of the program. When a catastrophic error occurs, a short (non-translated) message is written to unit 0, followed by a call to the C library routine **abort()**. A core dump can result, depending on how you configure your execution environment.

Severe errors

A severe error cannot be recovered from, even if the **ERR_RECOVERY** run-time option has been specified with the value **YES**. A severe error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement. If the

IOSTAT= and **ERR=** specifiers are not present on the input/output statement when a severe error condition is encountered, the program stops.

Table 25. IOSTAT Values for severe error conditions

IOSTAT Value	Error Description
1	END= is not specified on a direct access READ and the record is nonexistent.
2	End of file encountered on WRITE of an internal file.
6	File cannot be found and STATUS='OLD' is specified on an OPEN statement.
10	Read error on direct file.
11	Write error on direct file.
12	Read error on sequential or stream file.
13	Write error on sequential or stream file.
14	Error opening file.
15	Permanent I/O error encountered on file.
37	Dynamic memory allocation failure - out of memory.
38	REWIND error.
39	ENDFILE error.
40	BACKSPACE error.
107	File exists and STATUS='NEW' was specified on an OPEN statement.
119	BACKSPACE statement attempted on unit connected to a tape device.
122	Incomplete record encountered during direct access READ.
130	ACTION='READWRITE' specified on an OPEN statement to connect a pipe.
135	The user program is making calls to an unsupported version of the XL Fortran run-time environment.
139	I/O operation not permitted on the unit because the file was not opened with an appropriate value for the ACTION= specifier.
142	CLOSE error.
144	error.
152	ACCESS='DIRECT' is specified on an OPEN statement for a file that can only be accessed sequentially.
153	POSITION='REWIND' or POSITION='APPEND' is specified on an OPEN statement and the file is a pipe.
156	Invalid value for RECL= specifier on an OPEN statement.

Table 25. IOSTAT Values for severe error conditions (continued)

IOSTAT Value	Error Description
159	External file input could not be flushed because the associated device is not seekable.
165	The record number of the next record that can be read or written is out of the range of the variable specified with the NEXTREC= specifier of the INQUIRE statement.
169	The asynchronous I/O statement cannot be completed because the unit is connected for synchronous I/O only.
172	The connection failed because the file does not allow asynchronous I/O.
173	An asynchronous READ statement was executed while asynchronous WRITE statements were pending for the same unit, or an asynchronous WRITE statement was executed while asynchronous READ statements were pending for the same unit.
174	The synchronous I/O statement cannot be completed because an earlier asynchronous I/O statement has not been completed.
175	The WAIT statement cannot be completed because the value of the ID= specifier is invalid.
176	The WAIT statement cannot be completed because the corresponding asynchronous I/O statement is in a different scoping unit.
178	The asynchronous direct WRITE statement for a record is not permitted because an earlier asynchronous direct WRITE statement for the same record has not been completed.
179	The I/O operation cannot be performed on the unit because there are still incomplete asynchronous I/O operations on the unit.
181	A file cannot be connected to a unit because multiple connections are allowed for synchronous I/O only.
182	Invalid value for UWIDTH= option. It must be set to either 32 or 64.
183	The maximum record length for the unit is out of the range of the scalar variable specified with the RECL= specifier in the INQUIRE statement.
184	The number of bytes of data transmitted is out of the range of the scalar variable specified with the SIZE= or NUM= specifier in the I/O statement.
185	A file cannot be connected to two units with different UWIDTH values.

Table 25. IOSTAT Values for severe error conditions (continued)

IOSTAT Value	Error Description
186	Unit numbers must be in the range 0 to 2,147,483,647.
192	The value of the file position is out of the range of the scalar variable specified with the POS= specifier in the INQUIRE statement.
193	The value of the file size is out of the range of the scalar variable specified with the SIZE= specifier in the INQUIRE statement.
200	**FLUSH** error.
201	The unit specified in the **FLUSH** statement is connected to a non-seekable file.

Recoverable errors

A recoverable error is an error that can be recovered from. A recoverable error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement and the **ERR_RECOVERY** run-time option is set to **YES**, recovery action occurs and the program continues. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement and the **ERR_RECOVERY** option is set to **NO**, the program stops.

Table 26. IOSTAT values for recoverable error conditions

IOSTAT Value	Error Description
16	Value of REC= specifier invalid on direct I/O.
17	I/O statement not allowed on direct file.
18	Direct I/O statement on an unconnected unit.
19	Unformatted I/O attempted on formatted file.
20	Formatted I/O attempted on unformatted file.
21	Sequential or stream I/O attempted on direct file.
22	Direct I/O attempted on sequential or stream file.
23	Attempt to connect a file that is already connected to another unit.
24	OPEN specifiers do not match the connected file's attributes.
25	RECL= specifier omitted on an OPEN statement for a direct file.
26	RECL= specifier on an OPEN statement is negative.
27	ACCESS= specifier on an OPEN statement is invalid.

Table 26. IOSTAT values for recoverable error conditions (continued)

IOSTAT Value	Error Description
28	FORM= specifier on an OPEN statement is invalid.
29	STATUS= specifier on an OPEN statement is invalid.
30	BLANK= specifier on an OPEN statement is invalid.
31	FILE= specifier on an OPEN or INQUIRE statement is invalid.
32	STATUS='SCRATCH' and FILE= specifier specified on same OPEN statement.
33	STATUS='KEEP' specified on CLOSE statement when file was opened with STATUS='SCRATCH'.
34	Value of STATUS= specifier on CLOSE statement is invalid.
36	Invalid unit number specified in an I/O statement.
47	A namelist input item was specified with one or more components of nonzero rank.
48	A namelist input item specified a zero-sized array.
58	Format specification error.
93	I/O statement not allowed on error unit (unit 0).
110	Illegal edit descriptor used with a data item in formatted I/O.
120	The NLWIDTH setting exceeds the length of a record.
125	BLANK= specifier given on an OPEN statement for an unformatted file.
127	POSITION= specifier given on an OPEN statement for a direct file.
128	POSITION= specifier value on an OPEN statement is invalid.
129	ACTION= specifier value on an OPEN statement is invalid.
131	DELIM= specifier given on an OPEN statement for an unformatted file.
132	DELIM= specifier value on an OPEN statement is invalid.
133	PAD= specifier given on an OPEN statement for an unformatted file.
134	PAD= specifier value on an OPEN statement is invalid.
136	ADVANCE= specifier value on a READ statement is invalid.

Table 26. IOSTAT values for recoverable error conditions (continued)

IOSTAT Value	Error Description
137	ADVANCE='NO' is not specified when SIZE= is specified on a READ statement.
138	ADVANCE='NO' is not specified when EOR= is specified on a READ statement.
145	READ or WRITE attempted when file is positioned after the endfile record.
163	Multiple connections to a file located on a non-random access device are not allowed.
164	Multiple connections with ACTION='WRITE' or ACTION='READWRITE' are not allowed.
170	ASYNCH= specifier value on an OPEN statement is invalid.
171	ASYNCH= specifier given on an OPEN statement is invalid because the FORM= specifier is set to FORMATTED.
177	The unit was closed while there were still incomplete asynchronous I/O operations.
191	The RECL= specifier is specified on an OPEN statement that has ACCESS='STREAM'.
194	The BACKSPACE statement specifies a unit connected for unformatted stream I/O.
195	POS= specifier on an I/O statement is less than one.
196 [1]	The stream I/O statement cannot be performed on the unit because the unit is not connected for stream access.
197	POS= specifier on an I/O statement for a unit connected to a non-seekable file.
198	Stream I/O statement on an unconnected unit.
202 [1]	The ID=, POS=, or REC= specifier is not allowed in a child READ or WRITE statement.
203 [1]	The child READ or WRITE statement specified a unit number which does not match the unit number of the parent statement.
204 [1]	The child READ or WRITE statement is not allowed because the parent statement is not a READ or WRITE statement.
205	The user-defined derived type I/O procedure set the IOSTAT variable, but the parent statement did not specify IOSTAT=.
209	The BLANK= specifier in the READ statement has an illegal value.
210	A specifier in the READ statement has an illegal value.

Table 26. IOSTAT values for recoverable error conditions (continued)

IOSTAT Value	Error Description
211	The DELIM= specifier in the WRITE statement has an illegal value.
212 **1**	The data item in the formatted READ or WRITE statement must be processed by a DT edit descriptor. The READ or WRITE statement is ignored.
213	The NAMELIST item name encountered by the NAMELIST READ statement was not followed by an equals ('=')
214	The DELIM= specifier in the internal WRITE statement has an illegal value.
215	SIGN= specifier value on a WRITE statement is invalid for the external file.
216	SIGN= specifier value on a WRITE statement is invalid for the internal file.
217	SIGN= specifier given on an OPEN statement for an unformatted file.
218	SIGN= specifier value on an OPEN statement is invalid.
219	DECIMAL= specifier value is invalid for external file.
220	DECIMAL= specifier value is invalid for internal file.
221	DECIMAL= specifier is used in an unformatted I/O statement.
222	The ROUND= specifier was specified in an OPEN statement with FORM='UNFORMATTED'
223	The ROUND= specifier in the I/O statement has an illegal value.
224	There is no outstanding asynchronous data transfer specified by the ID= specifier.
225	A specifier in the OPEN statement has an illegal value.
226	There is no outstanding asynchronous data transfer specified.
227	Asynchronous data transfer error is not associated with the specified unit.
228	The UFMT_BIGENDIAN option was specified for a unit connected for formatted I/O.
229	The v-list of the DT edit descriptor contains an unexpected character.
230	The v-list of the DT edit descriptor contains an unexpected non-printable character.
231	Asynchronous data transfer error is not associated with the specified file.
232	OpenMP thread number is not available.

Table 26. IOSTAT values for recoverable error conditions (continued)

IOSTAT Value	Error Description
233	BACKSPACE performed on a unit that does not have read access.
235	ENCODING= specifier is used in OPEN statement for an unformatted file.
236	ENCODING= specifier has incorrect value in the OPEN statement.
240 **2**	NEWUNIT= specifer in an OPEN statement is missing FILE= or STATUS= with value 'SCRATCH'.
242	Invalid value specified for the CONVERT= specifier. The program recovers by ignoring the CONVERT= specifier.
243	An OPEN statement containing a CONVERT= specifier connects to a formatted data file. The recovery is to ignore the CONVERT= specifier.

Note:

1. Fortran 2003

2. Fortran 2008

Conversion errors

A conversion error occurs as a result of invalid data or the incorrect length of data in a data transfer statement. A conversion error causes the **IOSTAT=** specifier to be set to one of the values defined below and the **ERR=** label to be branched to if these specifiers are present on the input/output statement and the **CNVERR** option is set to **YES**. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement, both the **CNVERR** option and the **ERR_RECOVERY** option are set to **YES**, recovery action is performed and the program continues. If the **IOSTAT=** and **ERR=** specifiers are not present on the input/output statement, the **CNVERR** option is set to **YES**, the **ERR_RECOVERY** option is set to **NO**, and the program stops. If **CNVERR** is set to **NO**, the **ERR=** label is never branched to but the **IOSTAT=** specifier may be set, as indicated below.

Table 27. IOSTAT values for conversion error conditions

IOSTAT Value	Error Description	IOSTAT set if CNVERR=NO
3	End of record encountered on an unformatted file.	no
4	End of record encountered on a formatted external file using advancing I/O.	no
5	End of record encountered on an internal file.	no
7	Incorrect format of list-directed input found in an external file.	yes
8	Incorrect format of list-directed input found in an internal file.	yes
9	List-directed or NAMELIST data item too long for the internal file.	yes
41	Valid logical input not found in external file.	no
42	Valid logical input not found in internal file.	no

Table 27. IOSTAT values for conversion error conditions (continued)

IOSTAT Value	Error Description	IOSTAT set if CNVERR=NO
43	Complex value expected using list-directed or NAMELIST input in external file but not found.	no
44	Complex value expected using list-directed or NAMELIST input in internal file but not found.	no
45	NAMELIST item name specified with unknown or invalid derived-type component name in NAMELIST input.	no
46	NAMELIST item name specified with an invalid substring range in NAMELIST input.	no
49	List-directed or namelist input contained an invalid delimited character string.	no
56	Invalid digit found in input for B, O or Z format edit descriptors.	no
84	NAMELIST group header not found in external file.	yes
85	NAMELIST group header not found in internal file.	yes
86	Invalid NAMELIST input value found in external file.	no
87	Invalid NAMELIST input value found in internal file.	no
88	Invalid name found in NAMELIST input.	no
90	Invalid character in NAMELIST group or item name in input.	no
91	Invalid NAMELIST input syntax.	no
92	Invalid subscript list for NAMELIST item in input.	no
94	Invalid repeat specifier for list-directed or NAMELIST input in external file.	no
95	Invalid repeat specifier for list-directed or NAMELIST input in internal file.	no
96	Integer overflow in input.	no
97	Invalid decimal digit found in input.	no
98	Input too long for B, O or Z format edit descriptors.	no
121	Output length of NAMELIST item name or NAMELIST group name is longer than the maximum record length or the output width specified by the NLWIDTH option.	yes

Fortran 90, 95, 2003, and 2008 standard language errors
Fortran 90 standard language errors

A Fortran 90 language error results from the use of XL Fortran extensions to the Fortran 90 language that cannot be detected at compile time. A Fortran 90 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **90STD** and the **ERR_RECOVERY** run-time option has either not been set or is set to **NO**. If both **LANGLVL=90STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL= EXTENDED** is specified, the error condition is not considered an error.

Fortran 95 standard language errors

A Fortran 95 language error results from the use of XL Fortran extensions to the Fortran 95 language that cannot be detected at compile time. A Fortran 95 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **95STD** and the **ERR_RECOVERY** run-time option has either not been set or is set to **NO**. If both **LANGLVL=95STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL=EXTENDED** is specified, the error condition is not considered an error.

Fortran 2003 standard language errors

A Fortran 2003 standard language error results from the use of XL Fortran extensions to the Fortran 2003 language standard that cannot be detected at compile time. A Fortran 2003 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **2003STD** and the **ERR_RECOVERY** run-time option has either not been set or is set to **NO**. If both **LANGLVL=2003STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL=EXTENDED** is specified, the error condition is not considered an error.

Fortran 2008 standard language errors

A Fortran 2008 standard language error results from the use of XL Fortran extensions to the Fortran 2008 language standard that cannot be detected at compile time. A Fortran 2008 language error is considered a severe error when the **LANGLVL** run-time option has been specified with the value **2008STD** and the **ERR_RECOVERY** run-time option has either not been set or is set to **NO**. If both **LANGLVL=2008STD** and **ERR_RECOVERY=YES** have been specified, the error is considered a recoverable error. If **LANGLVL=EXTENDED** is specified, the error condition is not considered an error.

Table 28. IOSTAT Values for Fortran 90, 95, 2003, and 2008 Standard Language Error Conditions

IOSTAT Value	Error Description
53	Mismatched edit descriptor and item type in formatted I/O.
58	Format specification error.
140	Unit is not connected when the I/O statement is attempted. Only for READ, WRITE, PRINT, REWIND, and ENDFILE.
141	Two ENDFILE statements without an intervening REWIND or BACKSPACE on the unit.
151	The FILE= specifier is missing and the STATUS= specifier does not have a value of 'SCRATCH' on an OPEN statement.
187	NAMELIST comments are not allowed by the Fortran 90 standard.
199	STREAM is not a valid value for the ACCESS= specifier on an OPEN statement in Fortran 90 or Fortran 95.

Chapter 10. Input/Output formatting

Formatted **READ**, **WRITE** and **PRINT** data transfer statements use formatting information to direct the conversion between internal data representations and character representations in a formatted record. You can control the conversion process, called editing, by using a formatting type. The *Formatting and Access Types* table details the access types that support each formatting type.

Table 29. Formatting and access types

Formatting Type	Access Types
Format-directed	sequential, direct, and stream
List-directed	sequential and stream
Namelist	sequential and stream

Editing occurs on all fields in a record. A field is the part of a record that is read on input or written on output when format control processes a data or character string edit descriptor. The field width is the size of that field in characters.

Format-directed formatting

Format-directed formatting allows you to control editing using edit descriptors in a format specification. Specify a format specification in a **FORMAT** statement or as the value of a character array or character expression in a data transfer statement. Edit descriptors allow you to control editing in the following ways:

- Data edit descriptors allow you to specify editing by data type
- Control edit descriptors focus on the editing process
- Character string edit descriptors control string outputs

Complex editing

To edit complex values, you must specify complex editing by using a pair of data edit descriptors. A complex value is a pair of separate real components. When specifying complex editing, the first edit descriptor applies to the real part of the number. The second edit descriptor applies to the imaginary part of the number.

You can specify different edit descriptors for a complex editing pair and use one or more control edit descriptors between the edit descriptors in that pair. You must not specify data edit descriptors between the edit descriptors in that pair.

Data edit descriptors

Data edit descriptors allow you to specify editing by data type. You can use them to edit character, numeric, logical, and derived type data. The *Data Edit Descriptors* table contains a complete list of all character, character string, numeric, logical, and derived type edit descriptors. Numeric data refers to integer, real, and complex values.

Table 30. Data edit descriptors

Forms	Use
A **A***w*	Edits character values
B*w* **B***w.m*	Edits binary values
DT [1] **DT***char-literal-constant* [1] **DT***(v-list)* [1] **DT***char-literal-constant(v-list)* [1]	Edits an item of derived type. You can use a procedure instead of the default input/output formatting of an item of derived type.
E*w.d* **E***w.d***E***e* **E***w.d***D***e* [2] **E***w.d***Q***e* [2] **D***w.d* **EN***w.d* **EN***w.d***E***e* **ES***w.d* **ES***w.d***E***e* **Q***w.d* [2]	Edits real and complex numbers with exponents
F*w.d*	Edits real and complex numbers without exponents
G*w.d* **G***w.d***E***e* **G***w.d***D***e* [2] **G***w.d***Q***e* [2]	Edits data fields of any intrinsic type, with the output format adapting to the type of the data and, if the data is of type real, the magnitude of the data
I*w* **I***w.m*	Edits integer numbers
L*w*	Edits logical values
O*w* **O***w.m*	Edits octal values
Q [2]	Returns the count of characters remaining in an input record
Z*w* **Z***w.m*	Edits hexadecimal values

Notes:
  [1] Fortran 2003
  [2] IBM extension	

where:

char-literal-constant

> Specifies a character literal constant in a DT edit descriptor that must not have a kind parameter.

d   Specifies the number of digits to the right of the decimal point.

e	Specifies the number of digits in the exponent field.
m	Specifies the number of digits to print.
n	Specifies the number of characters in a literal field. Blanks are included in character count.

F2003 **v-list**
A comma-separated list of integer literal constants that have the same kind parameter. **F2003**

w	Specifies the width of a field including all blanks as a positive value.
	If you specify the **B**, **F**, **I**, **O**, or **Z**, edit descriptors on output, the value of *w* can be zero.

Rules for Data Edit Descriptor and Modifiers

You must not specify kind type parameters.

Edit descriptor modifiers must be unsigned integer literal constants.

IBM extension

For the *w*, *m*, *d*, and *e* modifiers, you must enclose a scalar integer expression in angle brackets (**<** and **>**). See "Variable format expressions (IBM extension)" on page 378 for details.

Note:

There are two types of **Q** data edit descriptor:

extended precision Q
is the **Q** edit descriptor with the**Q***w*.*d* syntax

character count Q
is the **Q** edit descriptor with the **Q** syntax

End of IBM extension

Rules for numeric edit descriptors on input

Leading blanks are not significant. You can control the interpretation of other blanks using the **BLANK=** specifier in the **OPEN** or **READ** statements and the **BN** and **BZ** edit descriptors. A field of all blanks is treated as zero.

Plus signs are optional, though you must not specify plus signs for the **B**, **O**, and **Z** edit descriptors.

In **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **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 contain more digits than can be represented internally.

Input of IEEE Exceptional Values

For real and complex editing, XL Fortran can now input IEEE exceptional values. The Fortran 2003 standard specifies a set of values for IEEE NaN (Not-a-Number) and IEEE infinity which XL Fortran now supports, along with another set of IEEE NaN values that are unique to XL Fortran. Input of IEEE exceptional values under real and complex editing are governed by the field width of the real or complex

edit descriptor. IEEE exceptional values are case insensitive during input. The **F**, **E**, **EN**, **ES**, **D**, **G**, and **Q** edit descriptors support the input of IEEE exceptional values.

The Fortran 2003 standard allows the following values for IEEE infinity: 'INF', '+INF', '-INF', 'INFINITY', '+INFINITY', or '-INFINITY'. These values can be preceded and followed by blanks.

The Fortran 2003 standard allows the following values for IEEE NaN: 'NAN', '+NAN', or '-NAN'. The sign that precedes 'NAN' will not have any significant meaning in XL Fortran. These values can also be preceded and followed by blanks. IEEE NaN can also be directly followed by zero or more characters in parentheses. The parentheses are used to indicate a quiet or signaling NaN. If only 'NAN' or 'NAN()' is specified it is interpreted as a quiet NaN. 'NAN(Q)' will be interpreted as a quiet NaN, and 'NAN(S)' as a signaling NaN. Any other alphanumeric characters specified inside the parentheses will have no significant meaning and will be interpreted as a quiet NaN by default.

As an IBM extension, XL Fortran allows the following values for IEEE NaN: 'NANQ' or 'NANS'. These exceptional values are case insensitive. 'NANQ' will be interpreted as a quiet NaN and 'NANS' as a signaling NaN. This form of IEEE NaN will only be allowed when the runtime option 'langlvl' is set to 'extended'.

Rules for numeric data edit descriptors on output

Characters are right-justified in the field.

When the number of characters in a field is less than the field width, leading blanks fill the remaining field space.

When the number of characters in a field is greater than the field width, or if an exponent exceeds its specified width, asterisks fill the entire field space.

A minus sign prefixes a negative value. A positive or zero value does not receive a plus sign prefix on output, unless you specify the **S**, **SP**, or **SS** edit descriptors.

If you specify the **-qxlf90** compiler option the **E**, **D**, **Q(Extended Precision)**, **F**, **EN**, **ES** and **G(General Editing)** edit descriptors output a negative value differently depending on the **signedzero** suboption.

- If you specify the **signedzero** suboption, the output field contains a minus sign for a negative value, even if that value is negative zero. This behavior conforms to the Fortran 95, Fortran 2003, and Fortran 2008 standards.

  IBM XL Fortran does not evaluate a **REAL(16)** internal value of zero as a negative zero. **IBM**

- If you specify the **nosignedzero** suboption, a minus sign is not written to the output field for a value of zero, even if the internal value is negative.

The **EN** and **ES** edit descriptors output a minus sign when the value is negative for the **signedzero** and **nosignedzero** suboptions.

Output of IEEE Exceptional Values

XL Fortran supports output of IEEE exceptional values for real and complex editing. Output of IEEE exceptional values can be Fortran 2003 standard compliant or compatible with previous releases of XL Fortran. A new compiler option and runtime option control the output of IEEE exceptional values. The **-qxlf2003=oldnaninf** compiler option will output IEEE exceptional values like

previous releases of XL Fortran; whereas, **-qxlf2003=nooldnaninf** will output IEEE exceptional values in accordance with the Fortran standard. In addition to the compiler option, a new runtime option, **naninfoutput**, can force the output of IEEE exceptional values to be Fortran 2003 standard compliant or compliant to the previous releases of XL Fortran. For more information on the **naninfoutput** runtime option see: Running XL Fortran programs section of the *XL Fortran Compiler Reference*. The **F**, **E**, **EN**, **ES**, **D**, **G**, and **Q** edit descriptors support the output of IEEE exceptional values.

Output of IEEE exceptional values under real and complex editing are governed by the field width of the real or complex edit descriptor. IEEE exceptional values are case sensitive during output.

Fortran 2003 Standard Output

IEEE infinity is output as 'Inf'. It can be preceded by as many blanks as necessary to be right justified. If the internal value is positive infinity, it can also be directly preceded by an optional plus sign if the field width allows for it. If the field width is less than three, asterisks are output instead. However, if the **SIGN=** specifier has a value of 'PLUS' or the 'sp' descriptor is used, then the plus sign is mandatory and the minimum field width is 4. If the internal value is negative infinity, it must be preceded by a negative sign. The minimum field width is 4. If the field width is less than four, asterisks are output instead.

IEEE Nan is output as 'NaN'. It can be preceded by as many blanks as necessary to be right justified. If the field width is greater than or equal to five, the standard allows for zero or more alphanumeric characters in parentheses to optionally follow the 'NaN'. XL Fortran will output 'NaN(Q)' for a quiet NaN and 'NaN(S)' for a signaling NaN if the field width is greater than five, otherwise only a 'NaN' is output. If the field width is less than three, asterisks are output instead.

Previous XL Fortran Output

IEEE infinity is output as 'INF'. It can be preceded by as many blanks as necessary to be right justified. If the field width is less than three, asterisks are output instead.

IEEE NaN is output as 'NaNQ' for a quiet NaN and 'NaNS' for a signaling NaN. It can also be directly preceded by an optional sign. It can be preceded by as many blanks as necessary to be right justified. If the field width is less than four, asterisks are output instead.

Rules for derived type edit descriptors (Fortran 2003)

The **DT** edit descriptor allows you to provide a procedure instead of the default input/output formatting for processing a list item of derived type. If you specify the optional *char-literal-constant*, the character value DT is concatenated to the *char-literal-constant* and passed to your user-defined derived-type input/output procedure as the iotype argument.

The values in the *v-list* of the **DT** edit descriptor are passed to the derived-type input/output procedure you define as the *v_list* array argument.

If a derived type variable or value corresponds to the **DT** edit descriptor, there must be an accessible interface to a derived type input/output procedure for that derived type.

You must not specify a DT edit descriptor as a non-derived type list item.

Control edit descriptors

Table 31. Control edit descriptors

Forms	Use
/ *r* /	Specifies the end of data transfer on the current record
:	Specifies the end of format control if there are no more items in the input/output list
$ **2**	Suppresses end-of-record in output *
BN	Ignores nonleading blanks in numeric input fields
BZ	Interprets nonleading blanks in numeric input fields as zeros
DC **1**	Specifies decimal comma as the decimal edit mode.
DP **1**	Specifies decimal point as the decimal edit mode.
*k***P**	Specifies a scale factor for real and complex items.
RU **1**	Specifies the **UP** rounding mode.
RC **1**	Specifies the **COMPATIBLE** rounding mode.
RD **1**	Specifies the **DOWN** rounding mode.
RN **1**	Specifies the **NEAREST** rounding mode.
RP **1**	Specifies the **PROCESSOR_DEFINED** rounding mode.
RZ **1**	Specifies the **ZERO** rounding mode.
S **SS**	Specifies that plus signs are not to be written
SP	Specifies that plus signs are to be written
T*c*	Specifies the absolute position in a record from which, or to which, the next character is transferred
TL*c*	Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred
TR*c* *o***X**	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred
Notes: **1** Fortran 2003 **2** IBM extension	

where:

r   is a repeat specifier. It is an unsigned, positive, integer literal constant.

k   specifies the scale factor to be used. It is an optionally signed, integer literal constant.

c   specifies the character position in a record. It is an unsigned, nonzero, integer literal constant.

o   is the relative character position in a record. It is an unsigned, nonzero, integer literal constant.

Rules for Control Edit Descriptors and Modifiers

You must not specify kind type parameters.

IBM *r*, *k*, *c*, and *o* can also be expressed as an arithmetic expression enclosed by angle brackets that evaluates into an integer value. **IBM**

Character string edit descriptors

Character string edit descriptors allow you to edit character data.

Forms	Use	Page
*n***H***str*	Outputs a character string (str)	"H Editing" on page 256
'*str*' "*str*"	Outputs a character string (*str*)	"Apostrophe/ Double quotation mark editing"

n   is the number of characters in a literal field. It is an unsigned, positive, integer literal constant. Blanks are included in character count. A kind type parameter cannot be specified.

Apostrophe/Double quotation mark editing
Purpose

The apostrophe/double quotation mark edit descriptor specifies a character literal constant in an output format specification.

Syntax

- '*character string*'
- "*character string*"

Rules

The width of the output field is the length of the character literal constant. See "Character" on page 44 for additional information on character literal constants.

──────────── **IBM extension** ────────────

Note:

1. A backslash is recognized, by default, as an escape sequence, and as a backslash character when the **-qnoescape** compiler option is specified. See escape sequences for more information.

2. XL Fortran provides support for multibyte characters within character constants, Hollerith constants, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.

3. Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

Examples

```
       ITIME=8
       WRITE(*,5) ITIME
5      FORMAT('The value is -- ',I2) ! The value is -- 8
       WRITE(*,10) ITIME
10     FORMAT(I2,'o''clock') ! 8o'clock
       WRITE(*,'(I2,7Ho''clock)') ITIME ! 8o'clock
       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
```

Effective list items (Fortran 2003)

This section discusses the rules for expanding a data transfer statement's array and derived-type input/output list items. The scalar objects that result from the application of these rules are called effective items. Zero-sized arrays and implied-**DO** lists with an iteration count of zero do not contribute to the effective list items. A scalar character item of zero length is an effective list item.

The following rules are re-applied to each expanded list item until none of the rules applies.

1. If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in array element order.

2. If a list item of derived type in an unformatted input/output statement is not processed by a user-defined derived-type input/output procedure, and if any subobject of that list item would be processed by a user-defined derived-type input/output procedure, the list item is treated as if all of the components of the object were specified in the list in component order. Those components are accessible in the scoping unit containing the input/output statement, and they must not be pointers or allocatable.

3. An effective input/output list item of derived type in an unformatted input/output statement is treated as a single value in a processor-dependent form, unless the list item or a subobject of a list item is processed by a user-defined derived-type input/output procedure.

4. If a list item of derived type in a *formatted* input/output statement is not processed by a user-defined derived-type input/output procedure, that list item is treated as if all of the components of the list item were specified in the list in component order. Those components are accessible in the scoping unit containing the input/output statement, and they must not be pointers or allocatable.

5. If a derived-type list item is not treated as a list of its individual components, its ultimate components cannot have the **POINTER** or **ALLOCATABLE** attribute, unless the list item is processed by a user-defined derived-type input/output procedure.

Interaction of Input/Output lists and format specifications

Beginning format-directed formatting initiates format control. Each action of format control depends on the next edit descriptor in the format specification, and on the next effective item in the input/output list, if one exists.

If an input/output list specifies at least one effective item, at least one data edit descriptor must exist in the format specification. Note that an empty format specification (parentheses only) can be used only if there are no effective items in the input/output list or if each item is a zero-sized array or an implied-**DO** list with an iteration count of zero. If this is the case and advancing input/output is in effect, one input record is skipped, or one output record containing no characters is written. For nonadvancing input/output, the file position is left unchanged.

A format specification is interpreted from left to right, except when a repeat specification (*r*) is present. A format item that is 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 effective item specified by the input/output list corresponds to each data edit descriptor. An effective list item of complex type requires the interpretation of two **F**, **E**, **EN**, **ES**, **D**, **G**, or extended precision **Q** edit descriptors. No item specified by the input/output list corresponds to a control edit descriptor or character string edit descriptor. Format control communicates information directly with the record.

Format control operates as follows:

1. If a data edit descriptor is encountered, format control processes an effective input/output list item, if there is one, or terminates the input/output command if the list is empty. If the effective list item processed is of type complex, any two edit descriptors are processed.

2. The colon edit descriptor terminates format control if no more effective items are in the input/output list. If more effective items are in the input/output list when the colon is encountered, it is ignored.

3. If the end of the format specification is reached, format control terminates if the entire effective input/output list has been processed, or control reverts to the beginning of the format item 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 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.

IBM During a read operation, any unprocessed characters of the record are skipped whenever the next record is read. A comma or semicolon can be used as a value separator for noncharacter data in an input record processed under format-directed formatting. The value separator will override the format width specifications when it appears before the end of the field width. For example, the format (I10,F20.10,I4) will read the following record correctly:

```
-345, .05E-3, 12
```

IBM

It is important to consider the maximum size record allowed on the input/output medium when defining a Fortran record by a **FORMAT** statement. For example, if a Fortran record is to be printed, the record should not be longer than the printer's line length.

Comma-separated Input/Output (IBM extension)

When reading floating-point data using format-directed input/output, a comma that appears in the input terminates the field. This can be useful for reading files containing comma-separated values.

For example, the following program reads two reals using the E edit descriptor. It requires that the field width be 16 characters. The program attempts to read the remaining characters in the record as a character string.

```
> cat read.f
real a,b
character*10 c
open(11, access='sequential', form='formatted')
read(11, '(2e16.10, A)') a,b,c
print *, a
print *, b
print *, c
end
```

If the floating-point fields are 16 characters wide, as the format specifies, the program executes correctly. (0.4000000000E+02 is 16 characters long.)

```
> cat fort.11
0.4000000000E+020.3000000000E+02hello
> a.out
 40.00000000
 30.00000000
 hello
```

But if the floating-point input contains less than 16 characters, errors occur because parts of the next field are read. (0.400000E+02 is 12 characters long.)

```
> cat fort.11
0.400000E+020.3000000E+02hello
> a.out
1525-097 A READ statement using decimal base input found the invalid digit
'.' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit
'h' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit
'e' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit
'l' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit
'l' in the input file.
The program will recover by assuming a zero in its place.
1525-097 A READ statement using decimal base input found the invalid digit
'o' in the input file.
The program will recover by assuming a zero in its place.
 INF
 0.0000000000E+00
```

If you use commas to terminate the fields, the floating-point values are read correctly. (0.400000E+02 is 12 characters long, but the fields are separated by commas.)

```
> cat fort.11
0.400000E+02,0.3000000E+02,hello
> a.out
 40.00000000
 30.00000000
 hello
```

If decimal comma mode is in effect, a semicolon acts as a value separator instead of a comma.

Data edit descriptors

In the examples of data edit descriptors, a lowercase b in the `Output` column indicates that a blank appears at that position.

A (Character) Editing
Purpose

The **A** edit descriptor directs the editing of character values. It can correspond to an input/output list item of type character or any other type. The kind type parameter of all characters transferred and converted is implied by the corresponding list item.

Syntax
- **A**
- **A***w*

Rules

On input, if *w* is greater than or equal to the length (call it *len*) of the input 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.

If *w* is not specified, the width of the character field is the length of the corresponding input/output list item.

`F2003` During formatted stream access, character output is split across more than one record if it contains newline characters. `F2003`

B (Binary) Editing
Purpose

The **B** edit descriptor directs editing between values of any type in internal form and their binary representation. (A binary digit is either 0 or 1.)

Syntax
- **B***w*
- **B***w.m*

Rules

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.

On input, w must be greater than zero.

On output, w can be zero. If w is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **B**w consists of zero or more leading blanks followed by the internal value in a form identical to the binary digits without leading zeros. Note that a binary constant always consists of at least one digit.

The output field for **B**$w.m$ is the same as for **B**w, except that the digit string consists of at least m digits. If necessary, the digit string is padded with leading zeros. The value of m must not exceed the value of w unless w is zero. If m is zero and the value of the internal data is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If m is zero, w is positive and the value of the internal datum is zero, the output field consists of w blank characters. If both w and m are zero, and the value of the internal datum is zero, the output field consists of only one blank character.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the entire output. On input, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors affect the **B** edit descriptor.

IBM If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **B**w is treated as **B**$w.m$, with m assuming the value that is the minimum of w and the number of digits required to represent the maximum possible value of the data item.
- The output consists of blanks followed by at least m digits. These are the rightmost digits of the number, zero-filled if necessary, until there are m digits. If the number is too large to fit into the output field, only the rightmost m digits are output.

If w is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors do not affect the **B** edit descriptor. **IBM**

Examples

Example 1: Examples of B editing on input

```
Input     Format     Value
111       B3         7
110       B3         6
```

Example 2: Examples of B editing on output

```
Value       Format          Output                Output
                        (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
  7           B3             111                    111
  6           B5             00110                  bb110
 17           B6.5           b10001                 b10001
 17           B4.2           0001                   ****
 22           B6.5           b10110                 b10110
 22           B4.2           0110                   ****
  0           B5.0           bbbbb                  bbbbb

  2           B0             10                     10
```

E, D, and Q (Extended Precision) Editing

Purpose

The **E**, **D**, and extended precision **Q** edit descriptors direct editing between real and complex numbers in internal form and their character representations with exponents. An **E**, **D**, or extended precision **Q** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, **IBM** or to any other type in XL Fortran, as long as the length is at least 4 bytes. **IBM**

Syntax

- **E***w*.*d*
- **E***w*.*d* **E***e*
- **D***w*.*d*
- **IBM** **E***w*.*d* **D***e* **IBM**
- **IBM** **E***w*.*d* **Q***e* **IBM**
- **IBM** **Q***w*.*d* **IBM**

Rules

The form of the input field is the same as for **F** editing. *e* has no effect on input.

The form of the output field for a scale factor of 0 is:
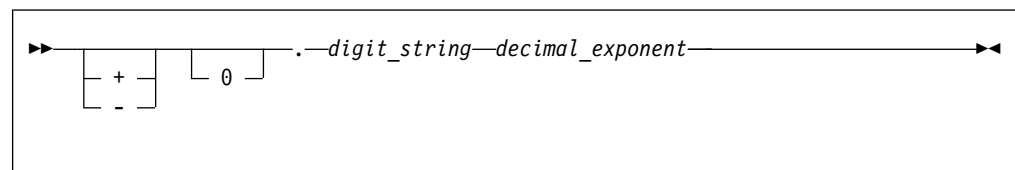
digit_string
>   is a digit string whose length is the *d* most significant digits of the value after rounding.

decimal_exponent
>   is a decimal exponent of one of the following forms (*z* is a digit):

Edit Descriptor	Absolute Value of Exponent (with scale factor of 0)	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| ≤ (10^e)-1	**E**±z_1z_2 ...z_e

Edit Descriptor	Absolute Value of Exponent (with scale factor of 0)	Form of Exponent
E$w.d$De *	|decimal_exponent| $\leq (10^e)-1$ *	D$\pm z_1 z_2 ... z_e$ *
E$w.d$Qe *	|decimal_exponent| $\leq (10^e)-1$ *	Q$\pm z_1 z_2 ... z_e$ *
D$w.d$	|decimal_exponent| ≤ 99	D$\pm z_1 z_2$
D$w.d$	$99<$|decimal_exponent| ≤ 309	$\pm z_1 z_2 z_3$
Q$w.d$ *	|decimal_exponent| ≤ 99 *	Q$\pm z_1 z_2$ *
Q$w.d$ *	$99<$|decimal_exponent| ≤ 309 *	$\pm z_1 z_2 z_3$ *

Note: * IBM extensions

The scale factor k (see "P (Scale Factor) Editing" on page 265) controls decimal normalization. If $-d<k\leq0$, the output field contains $|k|$ leading zeros and $d - |k|$ significant digits after the decimal symbol. If $0<k<d+2$, the output field contains k significant digits to the left of the decimal symbol and $d-k+1$ significant digits to the right of the decimal symbol. You cannot use other values of k.

For general information about numeric editing on input, see "Rules for numeric edit descriptors on input" on page 239.

For more information regarding numeric editing on output, see "Rules for numeric data edit descriptors on output" on page 240.

Examples

Example 1: Examples of E, D, and extended precision 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
```

Example 2: Examples of E, D, and extended precision Q editing on output

```
Value      Format         Output                  Output
                      (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
1234.56    E10.3          bb.123E+04              b0.123E+04
1234.56    D10.3          bb.123D+04              b0.123D+04
```

DT Editing (Fortran 2003)
Purpose

The DT edit descriptor allows you to specify that a user-defined procedure is called instead of the default input/output formatting for processing an input/output list item of derived type

Syntax
- **DT**
- **DT***char-literal-constant*
- **DT**(*v-list*)
- **DT***char-literal-constant*(*v-list*)

Rules

The **iotype** dummy argument passed to the user-defined input/output procedure contains the text from the *char-literal-constant*, prefixed with **DT**. If you do not include a *char-literal-constant*, the **iotype** argument contains only **DT**.

The *v-list* is passed to the user-defined input/output procedure in the *v_list* integer array dummy argument. If you do not include a *v-list*, the *v_list* dummy argument is a zero-sized array.

When you use the **DT** edit descriptor, the corresponding derived type input/output list item must be associated with an appropriate user-defined derived type input/output procedure.

EN Editing
Purpose

The **EN** edit descriptor produces an output field in the form of a real number in engineering notation such that the decimal exponent is divisible by 3 and the absolute value of the significand is greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The **EN** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, IBM or to any other type in XL Fortran, as long as the length is at least 4 bytes. IBM

Syntax

- **EN***w.d*
- **EN***w.d***E***e*

Rules

The form and interpretation of the input field is the same as for F editing.

The form of the output field is:
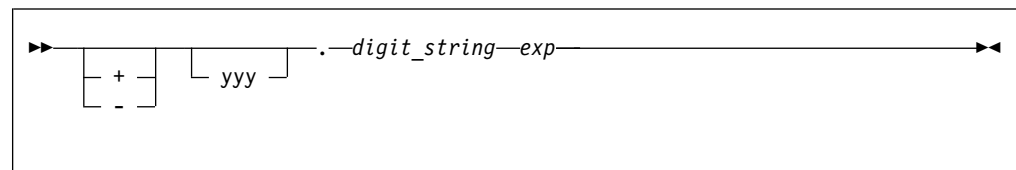
yyy   are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (*yyy* is an integer such that $1 \le yyy < 1000$ or, if the output value is zero, $yyy = 0$).

digit_string
         are the *d* next most significant digits of the value of the datum after rounding.

exp   is a decimal exponent, divisible by 3, of one of the following forms (z is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$|exp| \le 99$	$E\pm z_1 z_2$

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$99 < |exp| \le 309$	$\pm z_1 z_2 z_3$
EN*w.d*E*e*	$|exp| \le 10^e - 1$	$E \pm z_1 \ldots z_e$

For general information about numeric editing on input, see "Rules for numeric edit descriptors on input" on page 239.

For more information regarding numeric editing on output, see "Rules for numeric data edit descriptors on output" on page 240.

Examples

```
Value       Format      Output
3.14159     EN12.5      b3.14159E+00
1.41425D+5  EN15.5E4    141.42500E+0003
3.14159D-12 EN15.5E1    ***************

                        (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001      EN9.2       -1.00E-03                       -1.00E-03
```

ES Editing

Purpose

The **ES** edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

The **ES** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, **IBM** or to any other type in XL Fortran, as long as the length is at least 4 bytes. **IBM**

Syntax

- **ES***w.d*
- **ES***w.d*E*e*

Rules

The form and interpretation of the input field is the same as for F editing.

The form of the output field is:
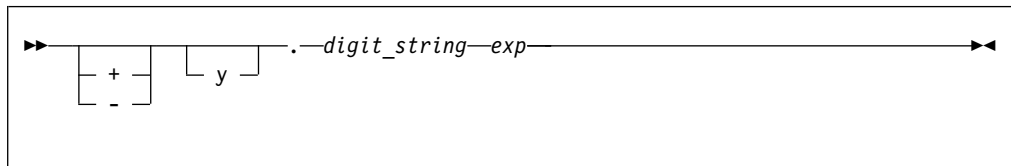
y   is a decimal digit representative of the most significant digit of the value of the datum after rounding.

digit_string
      are the *d* next most significant digits of the value of the datum after rounding.

exp   is a decimal exponent having one of the following forms (z is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
ES*w.d*	$\|exp\| \leq 99$	$E \pm z_1 z_2$
ES*w.d*	$99 < \|exp\| \leq 309$	$\pm z_1 z_2 z_3$
ES*w.d*E*e*	$\|exp\| \leq 10^e\text{-}1$	$E \pm z_1 \ldots z_e$

For general information about numeric editing on input, see "Rules for numeric edit descriptors on input" on page 239.

For more information regarding numeric editing on output, see "Rules for numeric data edit descriptors on output" on page 240.

Examples

```
Value       Format     Output
31415.9     ES12.5     b3.14159E+04
14142.5D+3  ES15.5E4   bb1.41425E+0007
31415.9D-22 ES15.5E1   ***************

                       (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001      ES9.2      -1.00E-03                   -1.00E-03
```

F (Real without Exponent) Editing
Purpose

The **F** edit descriptor directs editing between real and complex numbers in internal form and their character representations without exponents.

The **F** edit descriptor can correspond to an input/output list item of type real, to either part (real or imaginary) of an input/output list item of type complex, IBM▸ or to any other type in XL Fortran, as long as the length is at least 4 bytes. ◂IBM

Syntax

- **F***w.d*

Rules

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 symbol. If the decimal symbol is present, it overrides the *d* specified in the edit descriptor. If the decimal symbol is omitted, the rightmost *d* digits of the string are interpreted as following the decimal symbol, and leading blanks are converted to zeros if necessary.
3. Optionally, an exponent, having one of the following forms:
   - A signed digit string
   - **E**, **D**, or **Q** followed by zero or more blanks and by an optionally signed digit string. **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 symbol and represents the magnitude of the internal value, as modified by the scale factor in effect and rounded to d fractional digits. See "P (Scale Factor) Editing" on page 265 for more information.

On input, w must be greater than zero.

In Fortran 95 on output, w can be zero. If w is zero, the output field consists of the least number of characters required to represent the output value.

For general information about numeric editing on input, see "Rules for numeric edit descriptors on input" on page 239.

For more information regarding numeric editing on output, see "Rules for numeric data edit descriptors on output" on page 240.

Examples

Example 1: 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
```

Example 2: Examples of F editing on output

```
Value      Format          Output                      Output
                       (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
+1.2       F8.4            bb1.2000                    bb1.2000
.12345     F8.3            bbbb.123                    bbbb0.123
-12.34     F6.2            -12.34                      -12.34

-12.34     F0.2            -12.34                      -12.34


                       (with -qxlf90=signedzero)  (with -qxlf90=nosignedzero)
-0.001     F5.2            -0.00                        b0.00
```

G (General) Editing
Purpose

The **G** edit descriptor can correspond to an input/output list item of any type. Editing of integer data follows the rules of the **I** edit descriptor; editing of real and complex data follows the rules of the **E** or **F** edit descriptors (depending on the magnitude of the value); editing of logical data follows the rules of the **L** edit descriptor; and editing of character data follows the rules of the **A** edit descriptor.

Syntax
- **G**w.d
- **G**w.d**E**e
- IBM **G**w.d**D**e IBM
- IBM **G**w.d**Q**e IBM

Rules

For general information about numeric editing on input, see "Rules for numeric edit descriptors on input" on page 239.

For more information regarding numeric editing on output, see "Rules for numeric data edit descriptors on output" on page 240.

Examples

```
Value      Format        Output                Output
                     (with -qxlf77=gedit77)  (with -qxlf77=nogedit77)
0.0        G10.2        bb0.00E+00            bbb0.0
0.0995     G10.2        bb0.10E+00            bb0.10
99.5       G10.2        bb100.                bb0.10E+03
```

Generalized real and complex editing

If the **nogedit77** suboption (the default) of the **-qxlf77** option is specified, the method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If $0 < N < 0.1-0.5\times10^{-d-1}$ or $N \geq 10^{d}-0.5$ or N is 0 and d is 0, **G**$w.d$ output editing is the same as k**PE** $w.d$ output editing and **G**$w.d$**E**e output editing is the same as k**PE**$w.d$**E**e output editing, where k**P** refers to the scale factor ("P (Scale Factor) Editing" on page 265) currently in effect. If $0.1-0.5\times10^{-d-1} \leq N < 10^{d}-0.5$ or N is identically 0 and d is not zero, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
$N = 0$	F(w-n).(d-1),n('b') (d must not be 0)
$0.1-0.5\times10^{-d-1} \leq N < 1-0.5\times10^{-d}$	F(w-n).d,n('b')
$1-0.5\times10^{-d} \leq N < 10-0.5\times10^{-d+1}$	F(w-n).(d-1),n('b')
$10-0.5\times10^{-d+1}$ $\leq N < 100-0.5\times10^{-d+2}$	F(w-n).(d-2),n('b')
...	...
$10^{d-2}-0.5\times10^{-2} \leq N < 10^{d-1}-0.5\times10^{-1}$	F(w-n).1,n('b')
$10^{d-1}-0.5\times10^{-1} \leq N < 10^{d}-0.5$	F(w-n).0,n('b')

where b is a blank. n is 4 for **G**$w.d$ and $e+2$ for **G**$w.d$**E**e. The value of w-n must also be positive.

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

IBM If $0 < N < 0.1-0.5\times10^{-d-1}$, $N \geq 10^{d}-0.5$, or N is 0 and d is 0, **G**$w.d$**D**e output editing is the same as k**PE**$w.d$**D**e output editing and **G**$w.d$**Q**e output editing is the same as k**PE**$w.d$**Q**e output editing. **IBM**

On output, if the **gedit77** suboption of the **-qxlf77** compiler option is specified, 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 $N < 10^d$:

Magnitude of Datum	Equivalent Conversion
$0.1 \leq N < 1$	F$(w-n)$.d, $n('b')$
$1 \leq N < 10$	F$(w-n)$.(d-1), $n('b')$
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F$(w-n)$.1, $n('b')$
$10^{d-1} \leq N < 10^d$	F$(w-n)$.0, $n('b')$

Note: While FORTRAN 77 does not address how rounding of values affects the output field form, Fortran 90 does. Therefore, using **-qxlf77=gedit77** may produce a different output form than **-qxlf77=nogedit77** for certain combinations of values and **G** edit descriptors.

H Editing

Purpose

The **H** edit descriptor specifies a character string (*str*) and its length (*n*) in an output format specification. The string can consist of any of the characters allowed in a character literal constant.

Syntax

- *n***H** *str*

Rules

If an **H** edit descriptor occurs within a character literal constant, the constant delimiter character (for example, apostrophe) can be represented within *str* if two such characters are consecutive. Otherwise, another delimiter must be used.

The **H** edit descriptor must not be used on input.

Note: IBM

1. A backslash is recognized as an escape character by default, and as a backslash character when the **-qnoescape** compiler option is specified. See escape sequences for more information.

2. XL Fortran provides support for multibyte characters within character constants, Hollerith constants, character-string edit descriptors, and comments. This support is provided through the **-qmbcs** option. Assignment of a constant containing multibyte characters to a variable that is not large enough to hold the entire string may result in truncation within a multibyte character.

3. Support is also provided for Unicode characters and filenames. If the environment variable **LANG** is set to **UNIVERSAL** and the **-qmbcs** compiler option is specified, the compiler can read and write Unicode characters and filenames.

4. Fortran 95 does not include the **H** edit descriptor, although it was part of both FORTRAN 77 and Fortran 90. See page "Deleted features" on page 1054 for more information.

Examples

```
50    FORMAT(16HThe value is -- ,I2)
10    FORMAT(I2,7Ho'clock)
      WRITE(*,'(I2,7Ho''clock)') ITIME
```

I (Integer) Editing

Purpose

The **I** edit descriptor directs editing between integers in internal form and character representations of integers. The corresponding input/output list item can be of type integer IBM or any other type in XL Fortran. IBM

Syntax

- I*w*
- I*w*.*m*

Rules

w includes the optional sign.

m must have a value that is less than or equal to *w*, unless *w* is zero in Fortran 95.

The input field for the **I** edit descriptor must be an optionally signed digit string, unless it is all blanks. If it is all blanks, the input field is considered to be zeros.

m is useful on output only. It has no effect on input.

On input, *w* must be greater than zero.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

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:
   - A digit string without leading zeros if *m* is not specified
   - A digit string of at least *m* digits if *m* is specified and, if necessary, with leading zeros. If the internal value and *m* are both zero, blanks are written.

For additional information about numeric editing, see editing.

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero and the value of the internal datum is zero, the output field consists of only one blank character.

Examples

Example 1: Examples of I editing on input

(Assume **BN** editing is in effect for blank interpretation.)

```
Input      Format     Value
-123       I6         -123
123456     I7.5       123456
1234       I4         1234
```

Example 2: Examples of I editing on output

```
Value      Format     Output
-12        I7.6       -000012
12345      I5         12345

0          I6.0       bbbbbb
0          I0.0       b
2          I0         2
```

L (Logical) Editing
Purpose

The **L** edit descriptor directs editing between logical values in internal form and their character representations. The **L** edit descriptor can correspond to an input/output list item of type logical, IBM or any other type in XL Fortran. IBM

Syntax

- **L***w*

Rules

The input field consists of optional blanks, followed by an optional decimal symbol, followed by a T for true or an F for false. *w* includes blanks. Any characters following the T or F are accepted on input but are ignored; therefore, the strings .TRUE. and .FALSE. are acceptable input forms.

The output field consists of T or F preceded by (*w* - 1) blanks.

Examples

Example 1: Examples of L editing on input

```
Input      Format     Value
T          L4         true
 .FALSE.   L7          false
```

Example 2: Examples of L editing on output

```
Value      Format     Output
TRUE       L4         bbbT
FALSE      L1         F
```

O (Octal) Editing
Purpose

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.)

Syntax

- **O***w*
- **O***w.m*

Rules

w includes blanks.

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.

On input, *w* must be greater than zero.

On output, *w* can be zero. If *w* is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **O***w* consists of zero or more leading blanks followed by the internal value in a form identical to the octal digits without leading zeros. Note that an octal constant always consists of at least one digit.

The output field for **O***w.m* is the same as for **O***w*, except that the digit string consists of at least *m* digits. If necessary, the digit string is padded with leading zeros. The value of *m* must not exceed the value of *w*, unless *w* is zero. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the entire output. On input, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors affect the **O** edit descriptor.

IBM extension

If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **O***w* is treated as **O***w.m*, with *m* assuming the value that is the minimum of *w* and the number of digits required to represent the maximum possible value of the data item.
- The output consists of blanks followed by at least *m* digits. These are the rightmost digits of the number, zero-filled if necessary, until there are *m* digits. If the number is too large to fit into the output field, only the rightmost *m* digits are output.

If *w* is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors do not affect the **O** edit descriptor.

End of IBM extension

If *m* is zero, *w* is positive and the value of the internal datum is zero, the output field consists of *w* blank characters. If both *w* and *m* are zero and the value of the internal datum is zero, the output field consists of only one blank character.

Examples

Example 1: Examples of O editing on input

```
Input       Format      Value
123         O3          83
120         O3          80
```

Example 2: Examples of O editing on output

```
Value       Format          Output              Output
                        (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
80          O5              00120                   bb120
83          O2              23                      **

0           O5.0            bbbbb                   bbbbb
0           O0.0            b                       b
80          O0              120                     120
```

Q (Character Count) Editing (IBM extension)
Purpose

The character count **Q** edit descriptor returns the number of characters remaining in an input record. The result can be used to control the rest of the input.

Syntax

• **Q**

Rules

There also exists the extended precision **Q** edit descriptor. By default, XL Fortran only recognizes the extended precision **Q** edit descriptor described earlier. See "E, D, and Q (Extended Precision) Editing" on page 249 for more information. To enable both **Q** edit descriptors, you must specify the **-qqcount** compiler option.

When you specify the **-qqcount** compiler option, the compiler will distinguish between the two **Q** edit descriptors by the way the **Q** edit descriptor is used. If only a solitary **Q** is found, the compiler will interpret it as the character count **Q** edit descriptor. If **Q**w. or **Q**w.d is encountered, XL Fortran will interpret it as the extended precision **Q** edit descriptor. You should use correct format specifications with the proper separators to ensure that XL Fortran correctly interprets which **Q** edit descriptor you specified.

The value returned as a result of the character count **Q** edit descriptor depends on the length of the input record and on the current character position in that record. The value is returned into a scalar integer variable on the **READ** statement whose position corresponds to the position of the character count **Q** edit descriptor in the **FORMAT** statement.

The character count **Q** edit descriptor can read records of the following file types and access modes:
• Formatted sequential external files. A record of this file type is terminated by a new-line character. Records in the same file have different lengths.
• Formatted sequential internal nonarray files. The record length is the length of the scalar character variable.
• Formatted sequential internal array files. The record length is the length of an element in the character array.
• Formatted direct external files. The record length is the length specified by the **RECL=** specifier in the **OPEN** statement.
• Formatted stream external files. A record of this file type is terminated by a new-line character. Records in the same file have different lengths.

In an output operation, the character count **Q** edit descriptor is ignored. The corresponding output item is skipped.

Examples

```
@PROCESS QCOUNT
      CHARACTER(50) BUF
      INTEGER(4) NBYTES
      CHARACTER(60) STRING
      ...
      BUF = 'This string is 29 bytes long.'
      READ( BUF, FMT='(Q)' ) NBYTES
      WRITE( *,* ) NBYTES
! NBYTES equals 50 because the buffer BUF is 50 bytes long.
      READ(*,20) NBYTES, STRING
20    FORMAT(Q,A)
! NBYTES will equal the number of characters entered by the user.
      END
```

Z (Hexadecimal) Editing

Purpose

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, A-F, or a-f.)

Syntax

- **Z**w
- **Z**$w.m$

Rules

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.

On output, w can be zero. If w is zero, the output field consists of the least number of characters required to represent the output value.

The output field for **Z**w consists of zero or more leading blanks followed by the internal value in a form identical to the hexadecimal digits without leading zeros. Note that a hexadecimal constant always consists of at least one digit.

The output field for **Z**$w.m$ is the same as for **Z**w, except that the digit string consists of at least m digits. If necessary, the digit string is padded with leading zeros. The value of m must not exceed the value of w, unless w is zero. If m is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

If m is zero, w is positive and the value of the internal datum is zero, the output field consists of w blank characters.

If both w and m are zero and the value of the internal datum is zero, the output field consists of only one blank character.

If the **nooldboz** suboption of the **-qxlf77** compiler option is specified (the default), asterisks are printed when the output field width is not sufficient to contain the

entire output. On input, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors affect the **Z** edit descriptor.

IBM extension

If the **oldboz** suboption of the **-qxlf77** compiler option is specified, the following occurs on output:

- **Z**w is treated as **Z**$w.m$, with m assuming the value that is the minimum of w and the number of digits required to represent the maximum possible value of the data item.
- The output consists of blanks followed by at least m digits. These are the rightmost digits of the number, zero-filled if necessary, until there are m digits. If the number is too large to fit into the output field, only the rightmost m digits are output.

If w is zero, the **oldboz** suboption will be ignored.

With the **oldboz** suboption, the **BLANK=** specifier and the **BN** and **BZ** edit descriptors do not affect the **Z** edit descriptor.

End of IBM extension

Examples

Example 1: Examples of Z editing on input

```
Input      Format      Value
0C         Z2          12
7FFF       Z4          32767
```

Example 2: Examples of Z editing on output

```
Value      Format         Output              Output
                      (with -qxlf77=oldboz)  (with -qxlf77=nooldboz)
-1         Z2             FF                  **
12         Z4             000C                bbbC

12         Z0             C                   C
0          Z5.0           bbbbb               bbbbb
0          Z0.0           b                   b
```

Control edit descriptors

/ (Slash) Editing
Purpose

The slash edit descriptor indicates the end of data transfer on the current record. The repeat specifier (r) has a default value of 1.

Syntax

- /
- r/

Rules

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.

Fortran 2003

When you connect a file for input using stream access, each slash edit descriptor positions the file at the beginning of the next record, skipping the remaining portion of the current record. On output to a file connected for stream access, a newly created empty record follows the current record. The new record becomes both the current and last record of the file, with the file position coming at the beginning of the new record.

End of Fortran 2003

Examples
```
500   FORMAT(F6.2 / 2F6.2)
100   FORMAT(3/)
```

: (Colon) Editing
Purpose

The colon edit descriptor terminates format control if no more items are in the input/output list. If more items are in the input/output list when the colon is encountered, it is ignored.

Syntax

• :

Rules

See "Interaction of Input/Output lists and format specifications" on page 244 for more information.

Examples
```
10    FORMAT(3(:'Array Value',F10.5)/)
```

$ (Dollar) Editing (IBM extension)
Purpose

The dollar edit descriptor inhibits an end-of-record for a sequential or formatted stream **WRITE** statement.

Syntax

• $

Rules

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 writing to the same record.

Examples

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
```

BN (Blank Null) and BZ (Blank Zero) Editing
Purpose

The **BN** and **BZ** edit descriptors control the interpretation of nonleading blanks by subsequently processed **I**, **F**, **E**, **EN**, **ES**, **D**, **G**, **B**, **O**, **Z**, and extended precision **Q** edit descriptors. **BN** and **BZ** have effect only on input.

Syntax
- **BN**
- **BZ**

Rules

BN specifies that blanks in numeric input fields are to be ignored, and remaining characters are to be interpreted as though they were 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 **BLANK=** specifier of the **OPEN** statement. (See "OPEN" on page 416.) The initial setting is determined as follows:
- If **BLANK=** is not specified, blank interpretation is the same as if **BN** editing were specified.
- If **BLANK=** is specified, blank interpretation is the same as if **BN** editing were specified when the specifier value is **NULL**, or the same as if **BZ** editing were specified when the specifier value 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.

IBM If you specify the **oldboz** suboption of the **–qxlf77** compiler option, the **BN** and **BZ** edit descriptors do not affect data input edited with the **B**, **O**, or **Z** edit descriptors. Blanks are interpreted as zeros. **IBM**

DC and DP (Decimal) Editing (Fortran 2003)

Purpose

Decimal edit descriptors, **DC** and **DP** change the decimal edit mode to decimal comma and decimal point respectively.

Syntax

- **DC**
- **DP**

Rules

The decimal edit descriptors are used to control the representation of the decimal symbol in formatted input and output. The decimal comma or decimal point mode is in effect when the corresponding edit descriptor is encountered. They continue to be in effect until another **DC** or **DP** edit descriptor is encountered or until the end of the current I/O statement is reached.

Examples

```
program main
   real :: pi=3.14
   print '(t2, dp, f4.2)', pi
   print '(t2, dc, f4.2)', pi
end program

Output
3.14
3,14
```

P (Scale Factor) Editing

Purpose

The scale factor, k, applies to all subsequently processed **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **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. It is an optionally signed integer value representing a power of ten.

Syntax

- k**P**

Rules

On input, when an input field using an **F**, **E**, **EN**, **ES**, **D**, **G**, or extended precision **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 extended precision **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.

- In **EN** and **ES** editing, the scale factor has no effect.

Examples

Example 1: Examples of P editing on input

```
Input      Format      Value
98.765     3P,F8.6     .98765E-1
98.765     -3P,F8.6    98765.
.98765E+2  3P,F10.5    .98765E+2
```

Example 2: Examples of P editing on output

```
Value      Format          Output               Output
                       (with -qxlf77=noleadzero)  (with -qxlf77=leadzero)
5.67       -3P,F7.2        bbbb.01              bbb0.01
12.34      -2P,F6.4        b.1234               0.1234
12.34      2P,E10.3        b12.34E+00           b12.34E+00
```

RC, RD, RN, RP, RU, and RZ (Round) Editing (Fortran 2003)

Purpose

Round edit descriptors are used in a Format statement and are one of **RC**, **RD**, **RN**, **RP**, **RU**, and **RZ**, which correspond to the **COMPATIBLE**, **DOWN**, **NEAREST**, **PROCESSOR_DEFINED**, **UP**, and **ZERO** rounding modes respectively. The round edit descriptors temporarily change the connections rounding mode in formatted I/O. The round edit descriptors only affect **D**, **E**, **ES**, **EN**, **F** and **G** editing.

Syntax

- **RC**
- **RD**
- **RN**
- **RU**
- **RZ**

Rules

The round edit descriptors help specify how decimal numbers are converted to an internal representation (i.e. in binary) from a character representation and vice versa during formatted input and output.

Examples

```
program main
   real :: i
   100 format (f10.7, ru )
   open(UNIT=2,file ='temp.txt', form='formatted', round='compatible' )
   read(UNIT=2, 100) i
   print '(f10.7 , ru)' i
end program

Input - temp.txt
3.1415926
Output - temp.txt
3.1415928
```

S, SP, and SS (Sign Control) Editing
Purpose

The **S**, **SP**, and **SS** edit descriptors control the output of plus signs by all subsequently processed **I**, **F**, **E**, **EN**, **ES**, **D**, **G**, and extended precision **Q** edit descriptors until another **S**, **SP**, or **SS** edit descriptor is encountered or until format control terminates.

The sign control edit descriptors can temporarily overwrite the SIGN mode set by the SIGN=specifier for the connection. The **S**, **SP**, and **SS** edit descriptors set the sign mode corresponding to the SIGN=specifier values default, **PLUS** and **SUPPRESS**, respectively.

Syntax
- **S**
- **SP**
- **SS**

Rules

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.

Examples

```
Value      Format      Output
12.3456    S,F8.4      b12.3456
12.3456    SS,F8.4     b12.3456
12.3456    SP,F8.4     +12.3456
```

T, TL, TR, and X (Positional) Editing
Purpose

The **T**, **TL**, **TR**, and **X** edit descriptors specify the position where the transfer of the next character to or from a record starts.

Syntax
- **T***c*
- **TL***c*
- **TR***c*
- *o***X**

Rules

The **T** and **TL** edit descriptors use the left tab limit for file positioning. Immediately before the non-child data transfer the definition of the left tab limit is the character position of the current record or the current position of the stream file. The **T**, **TL**, **TR**, and **X** specify the character position as follows:
- For **T***c*, the *c*th character position of the record, relative to the left tab limit.
- For **TL***c*, *c* characters backward from the current position unless *c* is greater than the difference between the current character position and the left tab limit. Then, transmission of the next character to or from the record occurs at the left tab limit.
- For **TR***c*, *c* characters forward from the current position.

- For *o***X**, *o* 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.

IBM extension

The **X** edit descriptor can be specified without a character position. It is treated as 1X. When the source file is compiled with **-qlanglvl=90std** or **-qlanglvl=95std**, this extension is disabled in all compile-time format specifications, and the form of *o***X** is enforced. To disable this extension in run-time formats, the following run-time option must be set:

```
XLFRTEOPTS="langlvl=90std" or "langlvl=95std" ; export XLFRTEOPTS
```

End of IBM extension

Examples

Example 1: Examples of T, TL, and X editing on input

```
150    FORMAT(I4,T30,I4)
200    FORMAT(F6.2,5X,5(I4,TL4))
```

Example 2: 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')
```

List-directed formatting

List-directed formatting allows you to control the editing process using the lengths and types of data that is read or written. You can only use list-directed formatting with sequential or stream access.

Use the asterisk format identifier to specify list-directed formatting. For example:

```
REAL TOTAL1, TOTAL2
PRINT *, TOTAL1, TOTAL2
```

Value separators

If you specify list-directed formatting for a formatted record, that record consists of a sequence of values and value separators.

where:

value   is a constant or null.

value separator
> is a comma, slash, semicolon or set of adjacent blanks that occur between values in a record. You can specify one or more blanks before and after a comma or slash. If decimal comma mode is in effect, a semicolon replaces a comma as a value separator.

null is one of the following:

- 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.

> A null value has no effect on the definition status of the corresponding input list item.

List-directed input

Effective input list items in a list-directed **READ** statement are defined by corresponding values in a formatted record. The syntax of each value must agree with the type of the corresponding effective input list item.

Table 32. List-directed input

Syntax	Type
c	A literal constant of intrinsic type, or a non-delimited character constant.
r *	*r* is an unsigned, nonzero, integer literal constant. *r* * indicates *r* successive appearances of the null value.
r * *c*	Indicates *r* successive appearances of the constant.

Rules for list-directed input

You must not specify a kind type parameter for *c* or *r*.

List-directed formatting interprets two or more consecutive blanks as a single blank, unless the blanks are within a character value.

The constant *c* will have the same kind type parameter as the corresponding list item.

IBM Use the **-qintlog** compiler option to specify integer or logical values for input items of either integer or logical type. **IBM**

List-directed formatting interprets an object of derived type that occurs in an input list as if all structure components occur in the same order as in the derived type definition. The ultimate components of the derived type must not have the pointer **F2003** or allocatable **F2003** attribute.

A slash indicates the end of the input list and terminates list-directed formatting. Additional input list items after the slash evaluate as null values. If a slash is encountered by a child **READ** statement, it indicates the end of the input list for that particular child **READ** statement only. Any other input in the record following the slash is ignored. The slash has no effect on other child **READ** statements in the user-defined derived type I/O procedure or the parent **READ** statement.

Continuing a character value

A character value that meets the following conditions can continue in as many records as necessary:

- The next item or ultimate component of a derived type is of type character.
- The character constant does not contain the value separators blank, comma, or slash
- The character constant does not cross a record boundary.
- The first non-blank character is not a quotation mark or apostrophe.
- The leading characters are non numeric and followed by an asterisk.
- The character constant contains at least one character.

Delimiting apostrophes or quotation marks are not necessary to continue a character value across multiple records. If you omit delimiting characters, the first blank, comma, slash, or end-of-record terminates the character constant.

If you do not specify delimiting apostrophes or quotation marks, apostrophes and double quotation marks in the character value are not doubled.

End-of-record and list-directed input
In list-directed input an end-of-record has the same effect as a blank separator, unless the blank is within a character literal constant or complex literal constant. An end-of record does not insert a blank or any other character in a character value. An end-of-record must not occur between a doubled apostrophe in an apostrophe-delimited character sequence, or between a doubled quote in a quote-delimited character sequence

List-directed output

List-directed **PRINT** and **WRITE** statements output values in an order identical to the output list. Values are written in a form valid for the data type of each output list item.

Types of list-directed output

Table 33. List-directed output

Data Type	Form of Output
Arrays	Column-major order
Character	Depends on **DELIM=** specifier and file type, see Character Output.
Complex	Enclosed in parentheses with a comma separating the real and imaginary parts. Uses **E** or **F** editing.
Derived Types	User-defined derived-type I/O procedure.
Integer	Uses **I** editing.
Logical	T for a true value F for a false value
Real	Uses **E** or **F** editing.

List-directed character output
The output of character constants can change depending on the **DELIM=** specifier on the **OPEN** or **READ** statements.

Character constants output to a file opened without a **DELIM=** specifier, or a file opened with a **DELIM=** specifier with a value of NONE, output as follows:
- Values are not delimited by apostrophes or quotation marks.

- Value separators do not occur between values. Value separators will be emitted around the output of format-directed child I/O statements that have a list-directed parent statement.
- Each internal apostrophe or double quotation mark outputs as one apostrophe or double quotation mark.
- The processor inserts a blank character for carriage control at the beginning of any record that continues a character constant from the preceding record.

Note: Non-delimited character data can not always be read back correctly using list-directed input. Use with discretion.

Double quotation marks delimit character constants in a file opened with a **DELIM=** specifier with a value of QUOTE. A value separator follows the delimiter. Each internal quote outputs as two contiguous double quotation marks.

Apostrophes delimit character constants in a file opened with a **DELIM=** specifier with a value of APOSTROPHE A value separator follows the delimiter. Each internal apostrophe outputs as two contiguous apostrophes.

Rules for list-directed output

Each output record begins with a blank character that provides carriage control when that record outputs.

The end-of-record must not occur within a constant that is not character or complex.

In a complex constant, the end of a record can occur between the comma and the imaginary part of the constant only if the constant is as long or longer than a record. The only embedded blanks that can occur within a complex constant are one blank between the comma and the end of a record, and one blank at the beginning of the next record.

Blanks must not occur within a constant that is not character or complex.

Null values are not output.

Slashes you specify as value separators are not output.

─────────────────────────── **IBM extension** ───────────────────────────

For output that does not involve a user-defined derived-type I/O procedure, the *Width of a Written Field* table contains the width of the written field for any data type and length. The size of the record is the sum of the field widths plus one byte to separate each non-character field.

Table 34. Width of a written field

Data Type	Length (bytes)	Maximum Field Width (characters)	Fraction (decimal digits)	Precision/IEEE (decimal digits)
integer				
	1	4	n/a	n/a
	2	6	n/a	n/a
	4	11	n/a	n/a
	8	20	n/a	n/a

Table 34. Width of a written field (continued)

Data Type	Length (bytes)	Maximum Field Width (characters)	Fraction (decimal digits)	Precision/IEEE (decimal digits)
real	4	17	10	7
	8	26	18	15
	16	43	35	31
complex	8	37	10	7
	16	55	18	15
	32	89	35	31
logical	1	1	n/a	n/a
	2	1	n/a	n/a
	4	1	n/a	n/a
	8	1	n/a	n/a
character	n	n	n/a	n/a

End of IBM extension

Namelist formatting

Namelist formatting allows you to use the **NAME=** specifier as part of the **NAMELIST** statement to assign a name to a collection of variables. This name represents the entire collection of variables for input and output. You can also use namelist formatting to include namelist comments with input, making the data more user accessible.

- In Fortran 90 and Fortran 95, you can only use namelist formatting with sequential access.
- The Fortran 2003 standard allows you to use namelist formatting with sequential and stream access.
- The Fortran 2003 standard allows you to use namelist formatting with internal files.

Namelist input

The form of namelist input is:

1. Optional blanks and namelist comments.
2. The ampersand character, followed immediately by the namelist group name specified in the **NAMELIST** statement.
3. One or more blanks.
4. A sequence of zero or more name-value subsequences, separated by value separators.
5. A slash to terminate the namelist input.

Blanks at the beginning of an input record that continues a delimited character constant are considered part of the constant.

IBM If you specify the **NAMELIST=OLD** run-time option, the form of input for a **NAMELIST** statement is:

1. Optional blanks

2. An ampersand or dollar sign, followed immediately by the namelist group name specified in the **NAMELIST** statement.
3. One or more blanks.
4. A sequence of zero or more name-value subsequences separated by a single comma. You can insert a comma after the last name-value subsequence.
5. &END or $END to terminate the namelist input.

◀ IBM

The first character of each input record must be a blank, including those records that continue a delimited character constant.

Namelist comments

In Fortran 95 and higher, you can use comments in namelists.

F2003 ▶ You must not specify comments in stream input. ◀ F2003

If you specify the **NAMELIST=NEW** run-time option:
- If you specify an exclamation point after a value separator that is not a slash, or in the first non-blank position of a namelist input record, you initiate a comment. You can not initiate comments inside character literal constants.
- The comment extends to the end of the input record, and can contain any character in the XL Fortran character set.
- The comment is ignored.
- A slash within a namelist comment does not terminate execution of that namelist input statement.

IBM ▶ If you specify the **NAMELIST=OLD** run-time option:
- If you specify an exclamation point after a single comma or in the first non-blank position of a namelist input record that is not the first character of that record, you initiate a comment. You must not initiate a namelist comment within a character literal constant.
- The comment extends to the end of the input record, and can contain any character in the XL Fortran character set.
- The comment is ignored.
- An &END or $END within a namelist comment does not terminate execution of the namelist input statement.

◀ IBM

Name-value subsequence

The form of a name-value subsequence in an input record is:

```
>>--name-- = --constant_list--------------------------><
```

name   is a variable

constant
        has the following forms:

```
 ►►─┬──────┬──literal_constant──────────────────────────────────►◄
    └─r──*─┘
```
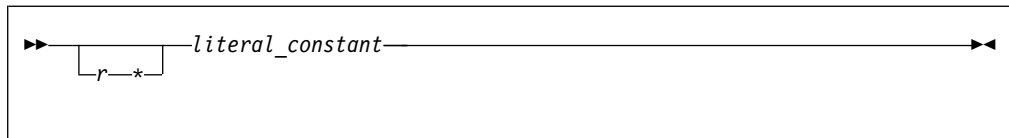
r   is an unsigned, nonzero, scalar, integer literal constant specifying the number of times the *literal_constant* occurs. You must not specify a kind type parameter for *r*.

literal_constant
          is a scalar literal constant of intrinsic type, or null value. You must not specify a kind type parameter for the constant. The constant evaluates with the same kind type parameter as the corresponding list item.

          You must specify delimiting apostrophes or quotation marks if *literal_constant* is of type character.

          You can specify T or F if *literal_constant* is of type logical.

Rules for namelist input

Any subscripts, strides, and substring range expressions that qualify *name* must be integer literal constants with no kind type parameter.

If *name* is not an array or an object of derived type, *constant_list* must contain a single constant.

Variable names you specify in the input file must appear in the *variable_name_list* of a **NAMELIST** statement. Variables can appear in any order.

If a name that you specify in an **EQUIVALENCE** statement shares storage with *name*, you must not substitute for that name in the *variable_name_list*.

You can use one or more optional blanks before or after *name*, but *name* must not contain embedded blanks.

In each name-value subsequence, the name must be the name of a namelist group item with an optional qualification. The name with the optional qualification must not be a:
*   zero-sized array.
*   zero-sized array section.
*   zero-length character string.

If you specify the optional qualification, it must not contain a vector subscript.

If *name* is an array, array section without vector subscripts, or a structure, *name* expands where applicable into a sequence of scalar list items of intrinsic data type according to the rules outlined in "Effective list items (Fortran 2003)" on page 244.

If *name* is an array or structure, the number of constants in *constant_list* must be less than or equal to the number of items specified by the expansion of *name*. If the number of constants is less than the number of items, the remaining items retain their former values.

You can specify a null value using:
*   The *r** form that indicates *r* successive appearances of the null value.

- Blanks between two consecutive value separators following an equal sign.
- Zero or more blanks preceding the first value separator and following an equal sign.
- Two consecutive non-blank value separators.

A null value has no effect on the definition status of the corresponding input list item. If the namelist group object list item is defined, it retains its previous value; if it is undefined, it remains undefined.

If decimal comma mode is in effect, a semicolon acts as a value separator instead of a comma.

You must not use a null value as the real or imaginary part of a complex constant. A single null value can represent an entire complex constant.

The end of a record following a value separator, with or without intervening blanks, does not specify a null value.

IBM extension

When you set the **LANGLVL=EXTENDED** run-time option, XL Fortran allows you to specify multiple input values in conjunction with a single array element. XL Fortran assigns the values to successive elements of that array, in array element order. The array element must not specify subobject designators.

Consider the following example, which declares array A as follows:

```
      INTEGER A(100)
      NAMELIST /FOO/ A
      READ (5, FOO)
```

Unit 5 contains the following input:

```
 &FOO
 A(3) = 2, 10, 15, 16
 /
```

During execution of the **READ** statement, XL Fortran assigns the following values:

- 2 to A(3)
- 10 to A(4)
- 15 to A(5)
- 16 to A(6)

If you specify multiple values in conjunction with a single array element, any logical constant must be specified with a leading period, for example, .T.

If you use the **NAMELIST=OLD** option at run time, the **BLANK=** specifier in the **OPEN** or **READ** statements determines how XL Fortran interprets embedded and trailing blanks between non-character constants.

If you specify the **-qmixed** compiler option, the namelist group name and list item names are case-sensitive.

End of IBM extension

A slash appearing as a value separator terminates the input statement after assignment of the previous value. Any additional items in the namelist group object receive null values

Example of namelist input data

File NMLEXP contains the following data before execution of the **READ** statement.

Character position:

```
          1         2         3
1...+....0....+....0....+....0
```

File contents:
```
 &NAME1
 I=5,
 SMITH%P_AGE=27
 /
```

NMLEXP contains four data records. The program contains the following code:
```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NAME1/ I,J,K,SMITH
I=1
J=2
K=3
SMITH=PERSON(20,'John Smith')
OPEN(7,FILE='NMLEXP')
READ(7,NML=NAME1)
! Only the value of I and P_AGE in SMITH are
! altered (I = 5, SMITH%P_AGE = 27).
! J, K and P_NAME in SMITH remain the same.
END
```

Note: In the previous example, data items appear in separate data records. The next example is a file with the same data items in one data record:

Character position:

```
          1         2         3         4
1...+....0....+....0....+....0....+....0
```

File contents:
```
 &NAME1 I= 5, SMITH%P_AGE=40 /
```

An example of a **NAMELIST** comment when you specify **NAMELIST=NEW**. The comment appears after the value separator space.
```
&TODAY I=12345         ! This is a comment. /
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0)/
```

IBM An example of a **NAMELIST** comment when you specify **NAMELIST=OLD**. The comment appears after the value separator space.
```
&TODAY I=12345,        ! This is a comment.
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0) &END
```

IBM

Namelist output

The **WRITE** statement outputs data from the *variable_name_list* in a **NAMELIST** statement according to data type. This data can be read using namelist input except for non-delimited character data.

You must not specify a single long character variable for namelist output.

Each output record that is not continuing a delimited character constant from a previous record begins with a blank character that provides carriage control.

The output data fields become large enough to contain all significant digits, as shown in the *Width of a Written Field* table.

The values of a complete array output in column-major order.

If the length of an array element is not sufficient to hold the data, you must specify an array with more than three elements.

IBM A **WRITE** statement with a *variable_name_list* produces a minimum of three output records:
- One record containing the namelist name.
- One or more records containing the output data items.
- One record containing a slash to terminate output.

To output namelist data to an internal file, the file must be a character array containing at least three elements. If you use the **WRITE** statement to transfer data to an internal file, the character array can require more than three elements. **IBM**

You can delimit character data using the **DELIM=** specifier on the **OPEN** or **READ** statements.

Namelist character output

The output of character constants can change depending on the **DELIM=** specifier on the **OPEN** or **READ** statements.

For character constants in a file opened without a **DELIM=** specifier, or with a **DELIM=NONE**:
- Values are non-delimited by apostrophes or quotation marks.
- Value separators do not occur between values.
- Each internal apostrophe or double quotation mark outputs as one apostrophe or quotation mark.
- XL Fortran inserts a blank character for carriage control at the beginning of any record that continues a character constant from the preceding record.

Nondelimited character data that has been written must not be read as character data.

Double quotation marks delimit character constants in a file opened with **DELIM=QUOTE**, with a value separator preceding and following each constant. Each internal quote outputs as two contiguous quotation marks.

Apostrophes delimit character constants in a file opened with **DELIM=APOSTROPHE** with a value separator preceding and following each constant. Each internal apostrophe outputs as two contiguous apostrophes.

Rules for namelist output

You must not specify a single character variable to output namelist data to an internal file, even if it is large enough to hold all of the data.

If you do not specify the **NAMELIST** run-time option, or you specify **NAMELIST=NEW**, the namelist group name and namelist item names output in uppercase.

IBM extension

If you specify **NAMELIST=OLD** at run-time:
- The namelist group name and namelist item names output in lower case.
- An &END terminates the output record.

If you specify **NAMELIST=OLD** at run-time and do not use the **DELIM=** specifier on an **OPEN** or **READ** statement:
- Apostrophes delimit character data
- Apostrophes delimit non-delimited character strings. A comma separator occurs between each character string.
- If a record starts with the continuation of a character string from the previous record, blanks are not added to the beginning of that record.

If you use the **-qmixed** compiler option, the namelist group name is case sensitive, regardless of the value of the **NAMELIST** run-time option.

To restrict namelist output records to a given width, use the **RECL=** specifier on the **OPEN** statement, or the **NLWIDTH** run-time option.

By default all output items for external files appear in a single output record. To have the record output on separate lines, use the **RECL=** specifier on the **OPEN** statement, or the **NLWIDTH** run-time option.

If decimal comma mode is in effect, a semicolon acts as a value separator instead of a comma.

End of IBM extension

Example of namelist output data

```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NL1/ I,J,C,SMITH
CHARACTER(5) :: C='BACON'
INTEGER I,J
I=12046
J=12047
SMITH=PERSON(20,'John Smith')
WRITE(6,NL1)
END
```

After execution of the **WRITE** statement with **NAMELIST=NEW**, the output data is:

```
      1         2         3         4
1...+....0....+....0....+....0....+....0
 &NL1
 I=12046, J=12047, C=BACON, SMITH=20, John Smith
 /
```

IBM▶ After execution of the **WRITE** statement with **NAMELIST=OLD**, the output data is:

```
      1         2         3         4
1...+....0....+....0....+....0....+....0
 &nl1
 i=12046, j=12047, c='BACON', smith=20, 'John Smith          '
 &end
```

◀IBM

Chapter 10. Input/Output formatting

Chapter 11. Statements and attributes

This section provides an alphabetical reference to all XL Fortran statements. The section for each statement is organized to help you readily access the syntax and rules, and points to the structure and uses of the statement.

The following table lists the statements, and shows which ones are executable, which ones are *specification_part* statements, and which ones can be used as the terminal statement of a **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct. The executable statements, specification statements, and terminal statements are marked with "√".

Table 35. Statements table

Statement Name	Executable Statement	Specification Statement	Terminal Statement
ABSTRACT **1**		√	
ALLOCATABLE **1**		√	
ALLOCATE	√		√
ASSIGN			√
ASSOCIATE **1**	√		
ASYNCHRONOUS **2**		√	
AUTOMATIC **2**		√	
BACKSPACE	√		√
BIND **1**		√	
BLOCK **3**			
BLOCK DATA			
BYTE **2**		√	
CALL	√		√
CASE	√		
CHARACTER		√	
CLASS **1**		√	
CLOSE	√		√
COMMON		√	
COMPLEX		√	
CONTAINS			
CONTIGUOUS **3**		√	
CONTINUE	√		√
CYCLE	√		
DATA		√	
DEALLOCATE	√		√
Derived Type			
DIMENSION		√	
DO	√		

Table 35. Statements table (continued)

Statement Name	Executable Statement	Specification Statement	Terminal Statement
DO CONCURRENT [3]	√		
DO WHILE	√		
DOUBLE COMPLEX [2]		√	
DOUBLE PRECISION		√	
ELSE	√		
ELSE IF	√		
ELSEWHERE	√		
END	√		
END ASSOCIATE [1]	√		
END BLOCK DATA			
END DO	√		√
END ENUM [1]		√	
END IF	√		
END FORALL	√		
END FUNCTION	√		
END INTERFACE		√	
END MAP [2]		√	
END MODULE			
END PROGRAM	√		
END SELECT	√		
END SUBMODULE [3]			
END SUBROUTINE	√		
END STRUCTURE [2]		√	
END TYPE		√	
END UNION [2]		√	
END WHERE	√		
ENDFILE	√		√
ENTRY		√	
ENUM [1]		√	
ENUMERATOR [1]		√	
EQUIVALENCE		√	
ERROR STOP [3]	√		
EXIT	√		
EXTERNAL		√	
FLUSH [1]	√		√
FORALL	√		√
FORMAT		√	
FUNCTION			
GO TO (Assigned)	√		

Table 35. Statements table (continued)

Statement Name	Executable Statement	Specification Statement	Terminal Statement
GO TO (Computed)	√		√
GO TO (Unconditional)	√		
IF (Arithmetic)	√		
IF (Block)	√		
IF (Logical)	√		√
IMPLICIT		√	
IMPORT **1**		√	
INQUIRE	√		√
INTEGER		√	
INTENT		√	
INTERFACE		√	
INTRINSIC		√	
LOGICAL		√	
MAP **2**		√	
MODULE			
MODULE PROCEDURE **3**			
NAMELIST		√	
NULLIFY	√		√
OPEN	√		√
OPTIONAL		√	
PARAMETER		√	
PAUSE	√		√
POINTER (Fortran 90)		√	
POINTER (integer) **2**		√	
PRINT	√		√
PRIVATE		√	
PROCEDURE **1**		√	
PROGRAM			
PROTECTED **1**		√	
PUBLIC		√	
READ	√		√
REAL		√	
RECORD **2**		√	
RETURN	√		
REWIND	√		√
SAVE		√	
SELECT CASE	√		
SELECT TYPE **1**	√		
SEQUENCE		√	

Table 35. Statements table (continued)

Statement Name	Executable Statement	Specification Statement	Terminal Statement
Statement Function		√	
STATIC **2**		√	
STOP	√		
SUBMODULE **3**			
SUBROUTINE			
STRUCTURE **2**		√	
TARGET		√	
TYPE		√	
Type Declaration		√	
Type Guard **1**	√		
UNION **2**		√	
USE		√	
VALUE **1**		√	
VECTOR **2**		√	
VIRTUAL **2**		√	
VOLATILE		√	
WAIT **1**	√		√
WHERE	√		√
WRITE	√		√

Notes:
 1 Fortran 2003
 2 IBM extension
 3 Fortran 2008

Assignment and pointer assignment statements are discussed in Chapter 6, "Expressions and assignment," on page 101. Both statements are executable and can serve as terminal statements.

Attributes

Each attribute has a corresponding attribute specification statement, and the syntax diagram provided for the attribute illustrates this form. An entity can also acquire this attribute from a type declaration statement or, in some cases, through a default setting. For example, entity *A*, said to have the **PRIVATE** attribute, could have acquired the attribute in any of the following ways:

```
REAL, PRIVATE :: A      ! Type declaration statement
PRIVATE :: A            ! Attribute specification statement

MODULE X
  PRIVATE               ! Default setting
  REAL :: A
END MODULE
```

ABSTRACT INTERFACE(Fortran 2003)

Purpose

The **ABSTRACT INTERFACE** statement is the first statement of an abstract interface block, which can specify procedure characteristics and dummy argument names without declaring a procedure with those characteristics.

Syntax

```
►►──ABSTRACT INTERFACE────────────────────────────────────────────────────►◄
```

Rules

If the interface statement is **ABSTRACT INTERFACE**, then the *function_name* in the function statement or the *subroutine-name* in the subroutine statement shall not be the same as a keyword that specifies an intrinsic type.

As an **ABSTRACT INTERFACE** cannot have a generic specification, a **PROCEDURE** statement is not allowed in the **ABSTRACT INTERFACE** block.

A *proc-language-binding-spec* with a **NAME=** specifier shall not be specified in the function or subroutine statement of an abstract interface body.

Examples

ABSTRACT INTERFACE can be used to declare the interface for deferred bindings.

```
ABSTRACT INTERFACE
 REAL FUNCTION PROC(A, B, C)
  REAL, INTENT (IN) :: A, B, C
 END FUNCTION
END INTERFACE
! P is declared to be a procedure pointer that is
! initially null with the same interfaceas procedure PROC.
PROCEDURE (PROC),POINTER:: P => NULL()
```

Procedure pointer P can point to any external procedure or module procedure if it has the same interface as *PROC*.

Related information
- "Abstract interface (Fortran 2003)" on page 175
- "INTERFACE" on page 405

ALLOCATABLE (Fortran 2003)

Purpose

The **ALLOCATABLE** attribute allows you to declare an allocatable object. You can dynamically allocate the storage space of these objects by executing an **ALLOCATE** statement or by a derived-type assignment statement. If the object is an array, it is a deferred-shape array or an assumed-rank array.

Syntax

```
>>--ALLOCATABLE-------------------------------------------------->
          |_::_|


      |-,<-------------------------------|
   >--+--object_name----------------------------------------------><
                    |--(--deferred_shape_spec_list--)--|
                    |_assumed_rank_spec_____|
```
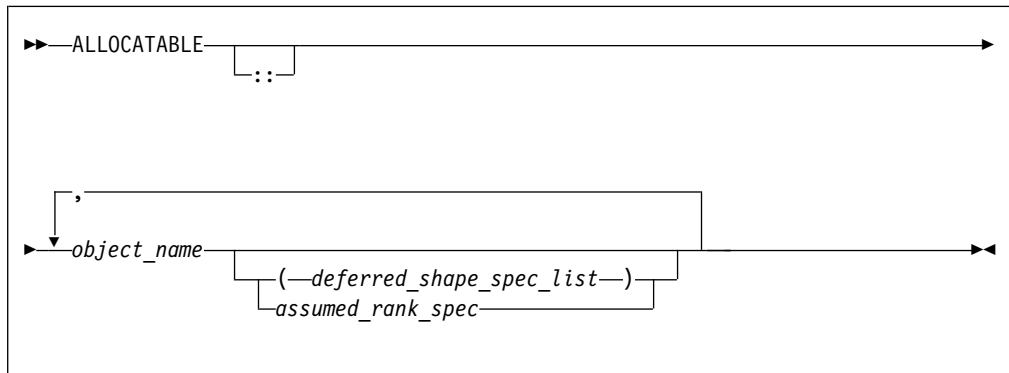
object_name
> The name of an allocatable object.

deferred_shape_spec
> A colon, where each colon represents a dimension.

assumed_rank_spec
> Two periods. For more information, see Assumed_rank_spec.

Rules

The object must not be a pointee.

If the object is an array specified elsewhere in the scoping unit with the **DIMENSION** attribute, the array specification must be a *deferred_shape_spec* or an *assumed_rank_spec*.

You can initialize an allocatable object after the storage space is allocated. If you compile your program with **-qinitalloc**, all uninitialized allocated objects are initialized.

Table 36. Attributes compatible with the ALLOCATABLE attribute

ASYNCHRONOUS	OPTIONAL	SAVE
AUTOMATIC **2**	PRIVATE	STATIC **2**
DIMENSION	PROTECTED **1**	TARGET
INTENT	PUBLIC	VOLATILE
Note: **1** Fortran 2003 **2** IBM extension		

Examples

```
! Two-dimensional array a declared but no space yet allocated
REAL, ALLOCATABLE :: a(:, :)
READ (5, *) i, j
ALLOCATE(a(i, j))
```

Related information

- "Allocatable arrays" on page 81
- "ALLOCATED(X)" on page 580

ALLOCATE

Purpose

The **ALLOCATE** statement dynamically provides storage for pointer targets and allocatable objects.

Syntax

```
►►──ALLOCATE──(──┬────────────────────────┬── allocation_list ──────────────────►
                 │                    (1) │              └─,──STAT──=──stat_variable─┘
                 └─ i_d_type_spec ──::──┘

►──┬──────────────────────────────────┬──┬──────────────────────────┬──)──►◄
   │                              (2)  │  │          (3)             │
   └─,──ERRMSG──=──errmsg_variable─────┘  └─,──┬──SOURCE─────┬──=──source_expr─┘
                                                │       (4)   │
                                                └──MOLD───────┘
```

Notes:

1   Fortran 2003

2   Fortran 2003

3   Fortran 2003
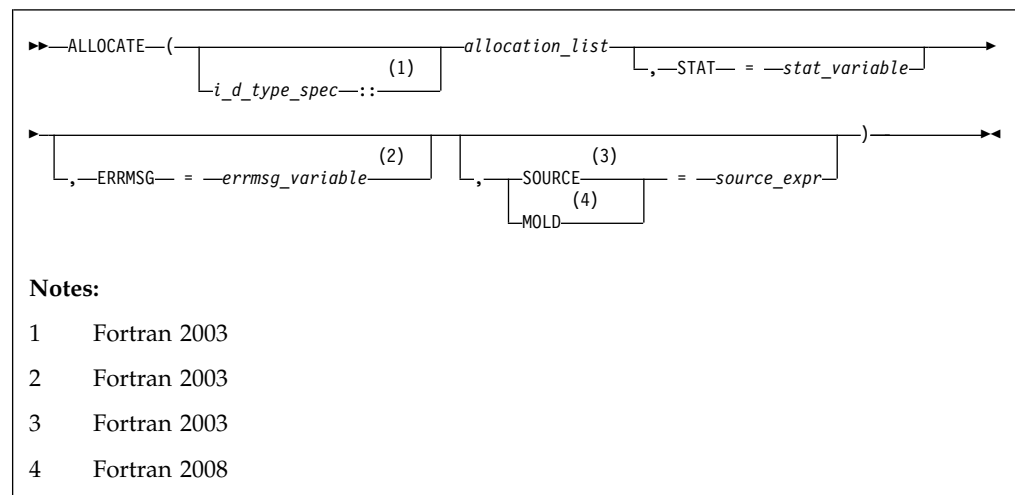
4   Fortran 2008

stat_variable
     A scalar integer variable.

errmsg_variable **(Fortran 2003)**
     A scalar character variable.

source_expr **(Fortran 2003)**
     An expression that is scalar or has the same rank as *allocate_object*.

i_d_type_spec **(Fortran 2003)**
     Is an *intrinsic_type_spec* or *derived_type_spec*. See "Type Declaration" on page 476 for a list of possible type specifications.

allocation_list
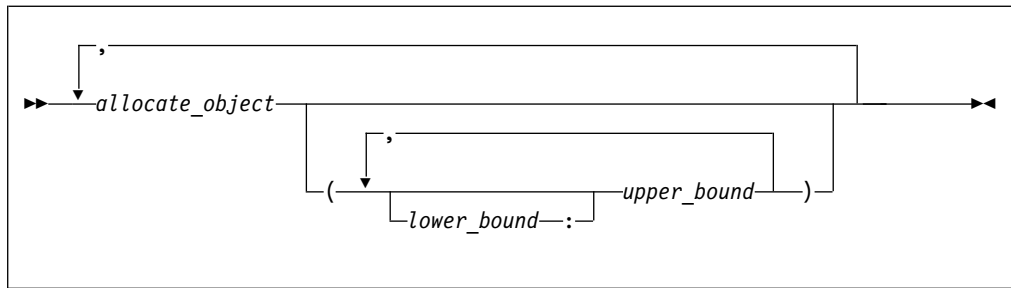
allocate_object
> A variable name or structure component that must be a data pointer or an allocatable object.

lower_bound
> A scalar integer expression.

upper_bound
> A scalar integer expression.

F2003

Constraints

You cannot specify *i_d_type_spec* and *source_expr* at the same time.

If any *allocate_object* has deferred type parameters, is unlimited polymorphic **F2008** , or is of an abstract type **F2008** , *i_d_type_spec* or *source_expr* must appear.

i_d_type_spec must specify a type with which each *allocate_object* is type compatible.

A type parameter value in *i_d_type_spec* must be an asterisk if and only if each *allocate_object* is a dummy argument whose corresponding type parameter is assumed.

The kind type parameter values of each *allocate_object* must be the same as the corresponding type parameter values of *i_d_type_spec*.

If *allocate_object* is an array, you must specify either the array bounds or *source_expr* with the same rank as *allocate_object*. If *allocate_object* is scalar, array bounds cannot be specified.

F2008 If *source_expr* appears, each *allocate_object* must be type-compatible with *source_expr*. **F2008**

Corresponding kind type parameters of *allocate_object* and *source_expr* must have the same values.

F2003

Rules

Execution of an **ALLOCATE** statement for a pointer causes the pointer to become associated with the target allocated. Execution of an **ALLOCATE** statement for an allocatable object causes the object to become definable.

If you specify bounds for an array, the number of dimensions specified (that is, the number of upper bounds in *allocation_list*) must be equal to the rank of *allocate_object*. **F2008** Otherwise, you must specify **SOURCE=** or **MOLD=**, and the bounds of *source_expr* determine the bounds of the array. Subsequent changes to the bounds of *source_expr* do not affect the array bounds after they are determined. **◄F2008**

F2003 If *i_d_type_spec* is specified, each *allocate_object* is allocated with the specified dynamic type and type parameter values. If *source_expr* is specified, each *allocate_object* is allocated with the dynamic type and type parameter values of *source_expr*. Otherwise, each *allocate_object* is allocated with the same dynamic type as its declared type.

When an **ALLOCATE** statement is executed for an array, the values of the bounds are determined at that time. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification. Any lower bound, if omitted, is assigned a default value of 1. If any lower bound value exceeds the corresponding upper bound value, that dimension has an extent of 0 and *allocate_object* is zero-sized.

Any *allocate_object* or a specified bound of an *allocate_object* must not depend on the value of *stat_variable* or *errmsg_variable*, or on the value, bounds, length type parameters, allocation status, or association status of any *allocate_object* in the same **ALLOCATE** statement.

stat_variable,*source_expr*, and *errmsg_variable* must not be allocated within the **ALLOCATE** statement in which they appear. They also must not depend on the value, bounds, length type parameters, allocation status, or association status of any *allocate_object* in the same **ALLOCATE** statement.**◄F2003**

F2008 If you specify **MOLD=** and *source_expr* is a variable, its value need not be defined. In addition, **MOLD=** does not copy the value of *source_expr* to the variable to be allocated. **◄F2008**

If the **STAT=** specifier is not present and an error condition occurs during execution of the statement, the program terminates. If the **STAT=** specifier is present, the *stat_variable* is assigned one of the following values:

IBM►

Stat value	Error condition
0	No error
1	Error in system routine attempting to do allocation
2	An invalid data object has been specified for allocation
3	Both error conditions 1 and 2 have occurred

◄IBM

F2003 If an error condition occurs during execution of the **ALLOCATE** statement and the *ERRMSG=specifier* is present, an explanatory message is assigned to *errmsg_variable*. If no such condition occurs, the value of *errmsg_variable* is not changed. **◄F2003**

Allocating an allocatable object that is already allocated causes an error condition in the **ALLOCATE** statement.

Pointer allocation creates an object that has the **TARGET** attribute. Additional pointers can be associated with this target (or a subobject of it) through pointer assignment. If you reallocate a pointer that is already associated with a target:
- A new target is created and the pointer becomes associated with this target.
- Any previous association with the pointer is broken.
- Any previous target that had been created by allocation and is not associated with any other pointers becomes inaccessible.

When an object of derived type is created by an **ALLOCATE** statement, any allocatable ultimate components have an allocation status of not allocated.

Use the **ALLOCATED** intrinsic function to determine if an allocatable object is allocated. Use the **ASSOCIATED** intrinsic function to determine the association status of a pointer or whether a pointer is associated with a specified target.

Examples

```
CHARACTER, POINTER :: P(:,:)
CHARACTER, TARGET :: C(4,4)
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
P => C
N = 2; M = N
ALLOCATE (P(N,M),STAT=I)         ! P is no longer associated with C
N = 3                            ! Target array for P maintains 2X2 shape
IF (.NOT.ALLOCATED(A)) ALLOCATE (A(N**2))
END
```

The following example uses the **MOLD=** specifier in an **ALLOCATE** statement in which the bounds are determined by reference to another object:

```
INTEGER, ALLOCATABLE :: NEW(:)
INTEGER, POINTER :: OLD(:)
ALLOCATE(OLD(4))
ALLOCATE (NEW, MOLD=OLD) ! Allocate NEW with the bounds of OLD
END
```

Related information

ASSIGN

Purpose

The **ASSIGN** statement assigns a statement label to an integer variable.

Syntax

```
>>--ASSIGN--stmt_label--TO--variable_name----------------------------------><
```

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

variable_name
> is the name of a scalar **INTEGER(4)** or **INTEGER(8)** variable

Rules

A statement containing the designated statement label must appear in the same scoping unit as the **ASSIGN** statement.

- If the statement containing the statement label is an executable statement, you can use the label name in an assigned **GO TO** statement that is in the same scoping unit.
- If the statement containing the statement label is a **FORMAT** statement, you can use the label name as the format specifier in a **READ**, **WRITE**, or **PRINT** statement that is in the same scoping unit.

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 before 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 the label itself, and you cannot use it as such.

The **ASSIGN** statement has been deleted from Fortran 95 and higher.

Examples

```
      ASSIGN 30 TO LABEL
      NUM = 40
      GO TO LABEL
      NUM = 50                 ! This statement is not executed
30    ASSIGN 1000 TO IFMT
      PRINT IFMT, NUM          ! IFMT is the format specifier
1000  FORMAT(1X,I4)
      END
```

Related information

- "Statement labels" on page 9
- "GO TO (assigned)" on page 383
- "Deleted features" on page 1054

ASSOCIATE (Fortran 2003)

Purpose

An **ASSOCIATE** statement is the first statement in an **ASSOCIATE** construct. It establishes an association between each identifier and either a variable or the value of an expression.

Syntax

```
>>--+---------------------------+--ASSOCIATE--(--+<---------------+--)--><
    |                           |                | ,            ↑ |
    +--associate_construct_name--:--+            +--associate_name--=>--selector--+
```
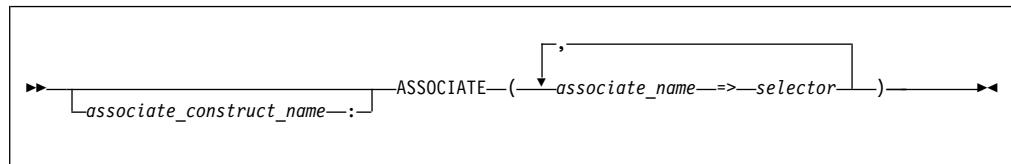
associate_construct_name
>is a name that identifies the **ASSOCIATE** construct

associate_name
>is an identifier that once associated with the selector, becomes an associating entity.

selector is a variable or expression that once associated becomes the associated entity.

Rules

If the *selector* is an expression or a variable with a vector subscript, the *associate_name* is assigned the value of the expression or variable. That associating entity must not become redefined or undefined.

If the *selector* is a variable without a vector subscript, the *associate_name* is associated with the data object specified by the *selector*. Whenever the value of the *associate_name* (or the associating entity identified by the *associate_name*) changes, the value of the variable changes with it.

If the *selector* has the **ALLOCATABLE** attribute, the associating entity does not have the **ALLOCATABLE** attribute. If the *selector* has the **POINTER** attribute, then the associating entity has the **TARGET** attribute. If the *selector* has the **TARGET**, **VOLATILE**, or F2003 **ASYNCHRONOUS** F2003 attribute, the associating entity that is a variable has those attributes.

If the *selector* has the **OPTIONAL** attribute, it must be present.

An associating entity has the same type, type parameters, and rank as the *selector*. If the *selector* is polymorphic, the associating entity is polymorphic. If the *selector* is an array, the associating entity is an array with a lower bound for each dimension equal to the value of the intrinsic **LBOUND**(*selector*). The upper bound for each dimension is equal to the lower bound plus the extent minus 1.

An *associate_name* must be unique within an **ASSOCIATE** construct.

If the *associate_construct_name* appears on an **ASSOCIATE** construct statement, it must also appear on the corresponding **END ASSOCIATE** statement.

An **ASSOCIATE** construct statement must not appear within the dynamic or lexical extent of a .

Examples

```
test_equiv: ASSOCIATE (a1 => 2, a2 => 40, a3 => 80)
  IF ((a1 * a2) .eq. a3) THEN
    PRINT *, "a3 = (a1 * a2)"
  END IF
END ASSOCIATE test_equiv

END
```

ASYNCHRONOUS

Purpose

The **ASYNCHRONOUS** statement specifies which variables may be associated with a pending I/O storage sequence while the scoping unit is in action.

[TS 29113] You must specify the **ASYNCHRONOUS** attribute for the variables that are used for asynchronous communication. See Interoperable variables in asynchronous communication for details. [TS 29113]

When you specify the **ASYNCHRONOUS** attribute for a variable, it is excluded from certain code motion optimizations.

Syntax

```
►►─ASYNCHRONOUS─::─ioitem_list──────────────────────────────────►◄
```

ioitem   is a variable name

Rules

The **ASYNCHRONOUS** attribute may be assigned implicitly by using a variable in an **ASYNCHRONOUS** I/O statement.

An object may have the **ASYNCHRONOUS** attribute in a particular scoping unit without having it in other scoping units.

If an object has the **ASYNCHRONOUS** attribute, then all of its subobjects have the **ASYNCHRONOUS** attribute.

An entity may have the **ASYNCHRONOUS** or **VOLATILE** attribute in the local scoping unit even if the associated module entity does not.

An accessed entity may have the **ASYNCHRONOUS** or **VOLATILE** attribute even if the host entity does not.

An associating entity has the **ASYNCHRONOUS** attribute if and only if the selector is a variable and has the **ASYNCHRONOUS** attribute.

Examples

Here is an example of variables that can be associated with a pending I/O sequence:

```
MODULE MOD
 INTEGER :: IOITEM
END MODULE

PROGRAM MAIN

 CALL SUB1()
 CALL SUB2()
END PROGRAM

SUBROUTINE SUB1() ! OPTIMIZATION MAY NOT BE PERFORMED
 USE MOD
 ASYNCHRONOUS :: IOITEM
 ....
END SUBROUTINE

SUBROUTINE SUB2() ! OPTIMIZATION MAY BE PERFORMED
 USE MOD
 .....
END SUBROUTINE

! OPTIMIZATION IS NOT POSSIBLE IN SUB2()IF MODULE MOD IS REWRITTEN AS FOLLOWS:

MODULE MOD
 INTEGER, ASYNCHRONOUS :: IOITEM
END MODULE
```

TS 29113

Here is an example of how to use the **ASYNCHRONOUS** attribute when doing asynchronous communication:

```
INTEGER :: ARRAY(1000)

... ! Code that writes to ARRAY

BLOCK
ASYNCHRONOUS :: ARRAY
  CALL MPI_ISEND(ARRAY,...REQ,...)
  ... ! Code that does not write to ARRAY
  CALL MPI_WAIT(REQ,...)
END BLOCK

... ! Code that writes to ARRAY
```

TS 29113

Related information
- "Asynchronous Input/Output" on page 220
- "OPEN" on page 416
- "CLOSE" on page 313
- "INQUIRE" on page 391
- "Interoperable variables in asynchronous communication (TS 29113)" on page 886
- "READ" on page 441
- "WAIT (Fortran 2003)" on page 491

AUTOMATIC (IBM extension)

Purpose

The **AUTOMATIC** attribute specifies that a variable has a storage class of automatic; that is, the variable is not defined once the procedure ends.

Syntax

```
►►──AUTOMATIC─────────────automatic_list──────────────────────────►◄
                └─::─┘
```
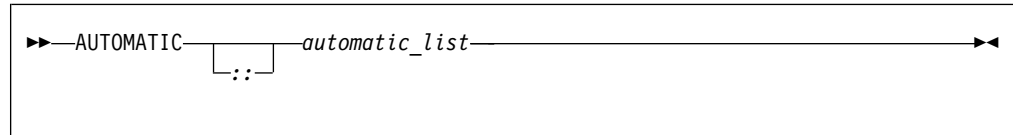
automatic
>   is a variable name or an array declarator with an explicit-shape specification list or a deferred-shape specification list

Rules

If *automatic* is a function result it must not be of type character or of derived type.

Function results that are pointers or arrays, dummy arguments, statement functions, automatic objects, or pointees must not have the **AUTOMATIC** attribute. A variable with the **AUTOMATIC** attribute cannot be defined in the scoping unit of a module `F2008` or submodule `F2008`. A variable that is explicitly declared with the **AUTOMATIC** attribute cannot be a common block item.

A variable must not have the **AUTOMATIC** attribute specified more than once in the same scoping unit.

Any variable declared as **AUTOMATIC** within the scope of a thread's work will be local to that thread.

A variable with the **AUTOMATIC** attribute cannot be initialized by a **DATA** statement or a type declaration statement.

If *automatic* is a pointer, the **AUTOMATIC** attribute applies to the pointer itself, not to any target that is (or may become) associated with the pointer.

Note: An object with the **AUTOMATIC** attribute should not be confused with an automatic object. See "Automatic objects" on page 20.

Table 37. Attributes compatible with the AUTOMATIC attribute

ALLOCATABLE **1**	CONTIGUOUS **2**	VOLATILE
ASYNCHRONOUS	POINTER	
DIMENSION	TARGET	
Note: **1** Fortran 2003 **2** Fortran 2008		

Examples

```
CALL SUB
CONTAINS
  SUBROUTINE SUB
    INTEGER, AUTOMATIC :: VAR
    VAR = 12
  END SUBROUTINE                 ! VAR becomes undefined
END
```

Related information

- "Storage classes for variables (IBM extension)" on page 28
- **-qinitauto** option in the *XL Fortran Compiler Reference*

BACKSPACE

Purpose

The **BACKSPACE** statement positions an external file connected for sequential access `F2003` or formatted stream access. `F2003`

`F2003` Execution of a **BACKSPACE** statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit. `F2003`

Syntax

```
>>--BACKSPACE--+--u-----------------+--><
               +--(--position_list--)--+
```

u
is an external unit identifier. The value of *u* must not be an asterisk or a Hollerith constant.

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:

[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 expression. The integer expression has one of the following values:

- A value in the range 1 through 2147483647
- `F2008` A NEWUNIT value `F2008`

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

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

IOMSG= *iomsg_variable* **(Fortran 2003)**
is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character

variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

is an input/output status specifier that specifies the status of the input/output operation. *ios* is an variable. When the **BACKSPACE** statement finishes executing, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

Rules

After the execution of a **BACKSPACE** statement, the file position is before the current record if a current record exists. If there is no current record, the file position is before the preceding record. If the file is at its initial point, file position remains unchanged.

You cannot backspace over records that were written using list-directed or namelist formatting.

For sequential access, if the preceding record is the endfile record, the file is positioned before the endfile record.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

IBM extension

If **IOSTAT=** and **ERR=** are not specified,

- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

End of IBM extension

Examples

```
   BACKSPACE 15
   BACKSPACE (UNIT=15,ERR=99)
      ...
99 PRINT *, "Unable to backspace file."
   END
```

Related information

- "Conditions and IOSTAT values" on page 226
- Chapter 9, "XL Fortran Input/Output," on page 215
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*

BIND (Fortran 2003)

Purpose

The **BIND** attribute declares that a Fortran variable or common block is interoperable with the C programming language.

Syntax

```
►►─BIND─(─C──────────────────────────────)───────┬──────┬──┬──variable_name─────┬──►◄
                │                        │        └─::─┘  │  └─common_block_name─┘
                └─,─NAME─ = ─binding_label─┘               (with repeat comma above)
```
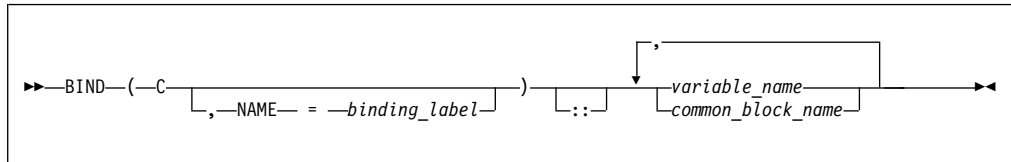
binding_label
>is a scalar character constant expression

Rules

This attribute specifies that a Fortran variable or common block is interoperable with a C entity with external linkage. Refer to "Interoperability of variables" on page 886 and "Interoperability of common blocks" on page 888 for details.

If the **NAME=** specifier appears in a **BIND** statement, then only one *variable_name* or *common_block_name* can appear.

If a **BIND** statement specifies a common block, then each variable of that common block must be of interoperable type and type parameters, and must not have the **POINTER** or **ALLOCATABLE** attribute.

Table 38. Attributes compatible with the BIND attribute

ASYNCHRONOUS	SAVE
DIMENSION	STATIC **2**
PRIVATE	TARGET
PROTECTED **1**	VOLATILE
PUBLIC	

Note:
  1 Fortran 2003
  2 IBM extension	

Related information
- Chapter 18, "Language interoperability features," on page 885
- "Interoperability of variables" on page 886
- "Interoperability of common blocks" on page 888
- "ENTRY" on page 358
- "FUNCTION" on page 379
- "PROCEDURE declaration (Fortran 2003)" on page 435
- "SUBROUTINE" on page 468
- "TYPE" on page 333
- –qbindcextname

BLOCK (Fortran 2008)

Purpose

The **BLOCK** statement declares a named or an unnamed **BLOCK** construct. It is the first statement of the **BLOCK** construct.

Syntax

```
►►─┬─────────────────────────┬─BLOCK──────────────────────────────────────────►◄
   └─BLOCK_construct_name──:──┘
```
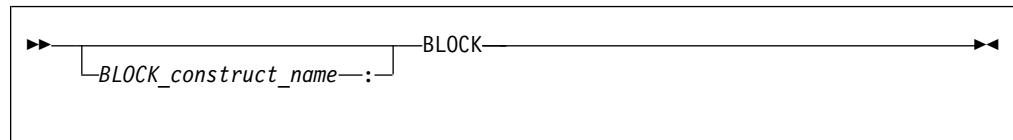
BLOCK_construct_name
> is a name that identifies the **BLOCK** construct.

Rules

If you specify a *BLOCK_construct_name* in a **BLOCK** statement, you must specify the same name in the corresponding **END BLOCK** statement.

Example

In the following example, the **BLOCK** statement declares an unnamed **BLOCK** construct:

```
SUBROUTINE swap(i, j)
  INTEGER :: i, j

  IF (i < j) THEN
    ! The BLOCK statement has no BLOCK_construct_name. The corrsponding END BLOCK
    ! statement cannot have a BLOCK_construct_name either.
    BLOCK
      INTEGER :: temp

      temp = i
      i = j
      j = temp
    END BLOCK
  END IF
END SUBROUTINE swap
```

Related information
- "BLOCK construct (Fortran 2008)" on page 136
- "END (Construct)" on page 351

BLOCK DATA

Purpose

A **BLOCK DATA** statement is the first statement in a block data program unit, which provides initial values for variables in named common blocks.

Syntax

```
▶▶──BLOCK DATA────────────────────────────────────────────────────▶◀
                  └─block_data_name─┘
```
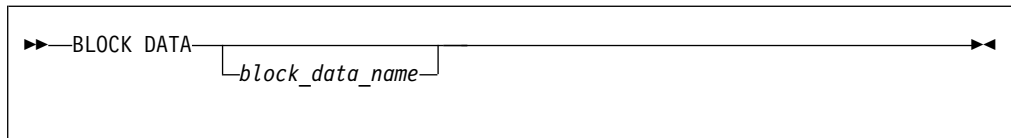
block_data_name
> is the name of a block data program unit

Rules

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

The name of the block data program unit, if given, must not be the same as an external subprogram, entry, main program, module, `F2008` submodule `F2008` , or common block in the executable program. It also must not be the same as a local entity in this program unit.

Examples

```
BLOCK DATA ABC
  PARAMETER (I=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*I/
END BLOCK DATA ABC
```

Related information

- "Block data program unit" on page 190
- "END" on page 350 for details on the **END BLOCK DATA** statement

BYTE (IBM extension)

Purpose

The **BYTE** type declaration statement specifies the attributes of objects and functions of type byte. Each scalar object has a length of 1. Initial values can be assigned to objects.

Syntax

```
▶▶──BYTE──┬──────────────────┬──entity_decl_list────────────────▶◀
          ├─::──────────────┤
          └─,──attr_spec_list─::─┘
```

where:

attr_spec
> is any of the following:

ALLOCATABLE **1**	INTRINSIC	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE

AUTOMATIC [2]	PARAMETER	STATIC [2]
BIND [1]	POINTER	TARGET
DIMENSION (*array_spec*)	PRIVATE	VALUE [1]
EXTERNAL	PROTECTED [1]	VOLATILE
INTENT (*intent_spec*)		
Note: [1] Fortran 2003 [2] IBM extension		

attr_spec
> For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

::
> is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
> is a list of dimension bounds

entity_decl

```
►►──a─┬──────────────────┬─┬────────────────────────────┬──►◄
       └─(─array_spec─)─┘  ├─/─initial_value_list─/──────┤
                           ├─=─constant_expr─────────────┤
                           └─ => ─NULL()─────────────────┘
```

a
> is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

initial_value
> provides an initial value for the entity specified by the immediately preceding name

constant_expr
> provides a constant expression for the entity specified by the immediately preceding name

=> **NULL()**
> provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module ►F2008 or submodule◄F2008 .

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If *constant_expr* or **NULL()** is specified, and the entity you are declaring:
- is a variable, the variable is initially defined.
- is a derived type component, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit.

A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the ►F2003 **ALLOCATABLE** or ◄F2003 **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
BYTE, DIMENSION(4) :: X=(/1,2,3,4/)
```

Related information

- "Byte (IBM extension)" on page 47
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

CALL

Purpose

The **CALL** statement invokes a subroutine to execute.

Syntax

```
►►─CALL─┬─name──────────────────────────────────┬─────────────►
        │                        (1)             │
        ├─procedure_component_ref────────────────┤
        │              (2)                        │
        └─data_ref──────separator─ binding_name──┘

►─┬───────────────────────────┬──────────────────────────────►◄
  └─(─┬─────────────────────┬─)─┘
      └─actual_argument_spec_list─┘
```

Notes:

1   Fortran 2003

2   Fortran 2003

name   The name of an internal, external, or module subroutine, an entry in an external or module subroutine, an intrinsic subroutine, a generic name, or a procedure pointer.

procedure_component_ref
        The name of a procedure pointer component of the declared type of *data_ref*. For details, see "Procedure pointer components" on page 53.

data_ref
        The name of an object of derived type

separator
        is % or IBM . IBM

binding_name
        is the name of a procedure binding of the declared type of *data_ref*

Rules

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.

Fortran 2003

A procedure pointer is a pointer that is associated with a procedure. Procedure pointers may have either an explicit or implicit interface and the interface may not be generic or elemental.

If the *binding_name* in a procedure designator is that of a *specific* procedure, the procedure referenced is the one identified by the binding with that name in the dynamic type of the *data_ref*. If the *binding_name* in a procedure designator is that of a *generic* procedure, the generic binding with that name in the declared type of the *data_ref* is used to select a specific binding according to the following rules:
1. If the reference is consistent with one of the specific bindings of that generic binding, that specific binding is selected.
2. Otherwise, if the reference is consistent with an elemental reference to one of the specific bindings of that generic binding, that specific binding is selected.

The reference is to the procedure identified by the binding with the same name as the selected specific binding, in the dynamic type of the *data_ref*.

End of Fortran 2003

A subprogram can call itself recursively, directly or indirectly, if the subroutine statement specifies the **RECURSIVE** keyword.

If a **CALL** statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statement labels indicated, depending on the action specified by the subroutine in the **RETURN** statement.

IBM extension

An external subprogram can also refer to itself directly or indirectly if the **-qrecur** compiler option is specified.

The argument list built-in functions **%VAL** and **%REF** are supplied to aid interlanguage calls by allowing arguments to be passed by value and by reference, respectively. They can only be references to non-Fortran procedures.

End of IBM extension

`F2003` The **VALUE** attribute also allows you to pass arguments by value. `F2003`

Examples

```
INTERFACE
  SUBROUTINE SUB3(D1,D2)
    REAL D1,D2
  END SUBROUTINE
```

```
        END INTERFACE
        ARG1=7 ; ARG2=8
        CALL SUB3(D2=ARG2,D1=ARG1)      ! subroutine call with argument keywords
        END

        SUBROUTINE SUB3(F1,F2)
          REAL F1,F2,F3,F4
          F3 = F1/F2
          F4 = F1-F2
          PRINT *, F3, F4
        END SUBROUTINE
```

Related information

- "Recursion" on page 208
- "%VAL and %REF (IBM extension)" on page 196
- "VALUE (Fortran 2003)" on page 487
- "Actual argument specification" on page 192
- "Asterisks as dummy arguments" on page 206
- "Type-bound procedures (Fortran 2003)" on page 60

CASE

Purpose

The **CASE** statement initiates a **CASE** statement block in a **CASE** construct, which has a concise syntax for selecting, at most, one of a number of statement blocks for execution.

Syntax
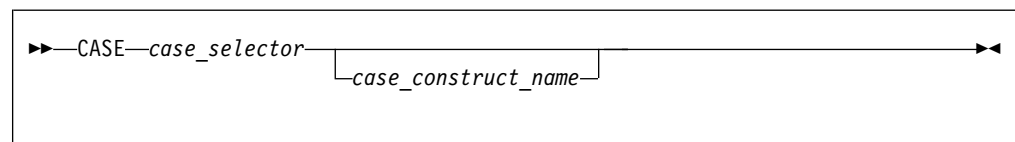
```
►►──CASE──case_selector──┬──────────────────────┬──────────────►◄
                         └─case_construct_name───┘
```

case_selector

```
►►──┬─DEFAULT────────────────────────────────────────────┬──►◄
    │              ┌─────,──────────┐                     │
    └─(──┬─▼─case_value────────────────────────┬──)──────┘
         ├──low_case_value──:──high_case_value──┤
         ├──low_case_value──:──────────────────┤
         └──:──high_case_value─────────────────┘
```
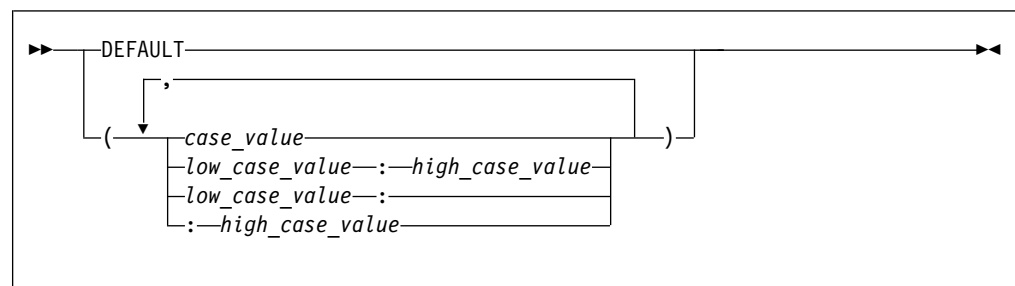
case_construct_name
> Is a name that identifies the **CASE** construct.

case_value
> is a scalar constant expression of type integer, character, or logical

low_case_value, high_case_value
> are each scalar constant expressions of type integer, character, or logical

Rules

The case index, determined by the **SELECT CASE** statement, is compared to each *case_selector* in a **CASE** statement. When a match occurs, the *stmt_block* associated with that **CASE** statement is executed. If no match occurs, no *stmt_block* is executed. No two case value ranges can overlap.

A match is determined as follows:

case_value

> DATA TYPE: integer, character or logical
> MATCH for integer and character: *case index = case_value*
> MATCH for logical: *case index* **.EQV.** *case_value* is
> true

low_case_value : high_case_value

> DATA TYPE: integer or character
> MATCH: *low_case_value* ≤ *case index* ≤
> *high_case_value*

low_case_value :

> DATA TYPE: integer or character
> MATCH: *low_case_value* ≤ *case index*

: high_case_value

> DATA TYPE: integer or character
> MATCH: *case index* ≤ *high_case_value*

DEFAULT

> DATA TYPE: not applicable
> MATCH: if no other match occurs.

There must be only one match. If there is a match, the statement block associated with the matched *case_selector* is executed, completing execution of the case construct. If there is no match, execution of the case construct is complete.

If the *case_construct_name* is specified, it must match the name specified on the **SELECT CASE** and **END SELECT** statements.

DEFAULT is the default *case_selector*. Only one of the **CASE** statements may have **DEFAULT** as the *case_selector*.

Each case value must be of the same data type as the *case_expr*, as defined in the **SELECT CASE** statement. If any typeless constants or **BYTE** named constants are encountered in the *case_selectors*, they are converted to the data type of the *case_expr*.

When the *case_expr* and the case values are of type character, they can have different lengths. If you specify the **-qctyplss** compiler option, a character constant expression used as the *case_expr* remains as type character. The character constant expression will not be treated as a typeless constant.

Examples

```
ZERO: SELECT CASE(N)

  CASE DEFAULT ZERO          ! Default CASE statement for
                             ! CASE construct ZERO
      OTHER: SELECT CASE(N)
         CASE(:-1)           ! CASE statement for CASE
                             ! construct OTHER

              SIGNUM = -1
         CASE(1:) OTHER
              SIGNUM = 1
      END SELECT OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
```

Related information

- "CASE construct" on page 145
- "SELECT CASE" on page 459
- "END (Construct)" on page 351, for details on the **END SELECT** statement

CHARACTER

Purpose

A **CHARACTER** type declaration statement specifies the kind, length, and attributes of objects and functions of type character. You can assign initial values to objects.

Syntax

where:

attr_spec
>   is any of the following:

ALLOCATABLE [1]	INTRINSIC	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE
AUTOMATIC [2]	PARAMETER	STATIC [2]
BIND [1]	POINTER	TARGET
DIMENSION (*array_spec*)	PRIVATE	VALUE [1]
EXTERNAL	PROTECTED [1]	VOLATILE
INTENT (*intent_spec*)		

char_selector

> specifies the character length.

> *type_param_value*
> > is a specification expression, an asterisk (*) or a colon
>
> *int_constant_expr*
> > is a scalar integer constant expression that must evaluate to 1
>
> *char_length*
> > is either a scalar integer literal constant (which cannot specify a
> > kind type parameter) or a *type_param_value* enclosed in parentheses

attr_spec
> For detailed information on rules about a particular attribute, refer to the
> statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

::   is the double colon separator. Use the double colon separator when you
> specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
> is a list of dimension bounds.

entity_decl

```
>>--a--------------------------------------------------------------->
        |  *  --char_length-- |    |--(--array_spec--)--|
        |--(--array_spec--)-- * --char_length------------|

>------------------------------------------------------------------><
    |                                          (1) |
    |--/--initial_value_list--/--------------------|
    |-- = --constant_expr--------------------------|
    |-- => --NULL()--------------------------------|
```
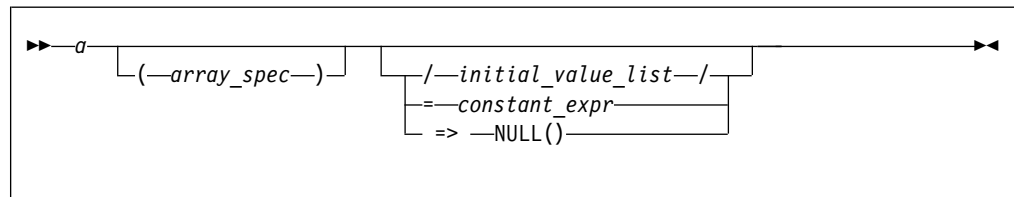
Notes:

1   IBM extension

a   is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

initial_value
         provides an initial value for the entity specified by the immediately preceding name.

constant_expr
         provides a constant expression for the entity specified by the immediately preceding name.

=> **NULL()**
         provides the initial value for the pointer object.

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements. For details, see "Type Declaration" on page 476.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object must not be initially defined in a type declaration statement if it is a dummy argument, an allocatable object, a pointer, a function result, an object in

blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if:

- it appears in a named common block in a block data program unit.
- if it appears in a named common block in a module ▶F2008 or submodule

   ◀F2008 .

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of a *type_param_value* or an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If *constant_expr* or **NULL()** is specified, and the entity you are declaring:

- is a variable, the variable is initially defined.
- is a derived type component, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit.

A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in an *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute. A *char_length* specified in an *entity_decl* takes precedence over any length specified in *char_selector*.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

▶IBM If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. ◀IBM

The optional comma after *char_length* in a **CHARACTER** type declaration statement is permitted only if no double colon separator (::) appears in the statement.

If the **CHARACTER** type declaration statement specifies a length of a colon, the
length type parameter is a *deferred type parameter*. An entity or component with a
deferred type parameter must specify the **ALLOCATABLE** or **POINTER** attribute.
A deferred type parameter is a length type parameter whose value can change
during the execution of the program.

If the **CHARACTER** type declaration statement is in the scope of a module,
`F2008` submodule `F2008` , block data program unit, or main program, and you
specify the length of the entity as an inherited length, the entity must be the name
of a named character constant. The character constant assumes the length of its
corresponding expression defined by the **PARAMETER** attribute.

If the **CHARACTER** type declaration statement is in the scope of a procedure and
the length of the entity is inherited, the entity name must be the name of a dummy
argument or a named character constant. If the statement is in the scope of an
external function, it can also be the function or entry name in a **FUNCTION** or
ENTRY statement in the same program unit. If the entity name is the name of a
dummy argument, the dummy argument assumes the length of the associated
actual argument for each reference to the procedure. If the entity name is the name
of a character constant, the character constant assumes the length of its
corresponding expression defined by the **PARAMETER** attribute. If the entity
name is a function or entry name, the entity assumes the length specified in the
calling scoping unit.

The length of a character function can be a specification expression (which must be
a constant expression if the function type is not declared in an interface block) or it
is a colon, or an asterisk, indicating the length of a dummy procedure name. The
length cannot be an asterisk if the function is an internal or module function, if it
is recursive, or if it returns array or pointer values.

Examples

```
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER(7), TARGET :: ORANGES = 'ORANGES'
I=7
CALL TEST(APPLES,I)
CONTAINS
  SUBROUTINE  TEST(VARBL,I)
    CHARACTER*(*), OPTIONAL :: VARBL    ! VARBL inherits a length of 6
    CHARACTER(I) :: RUNTIME             ! Automatic object with length of 7
  END SUBROUTINE
END
```

Related information

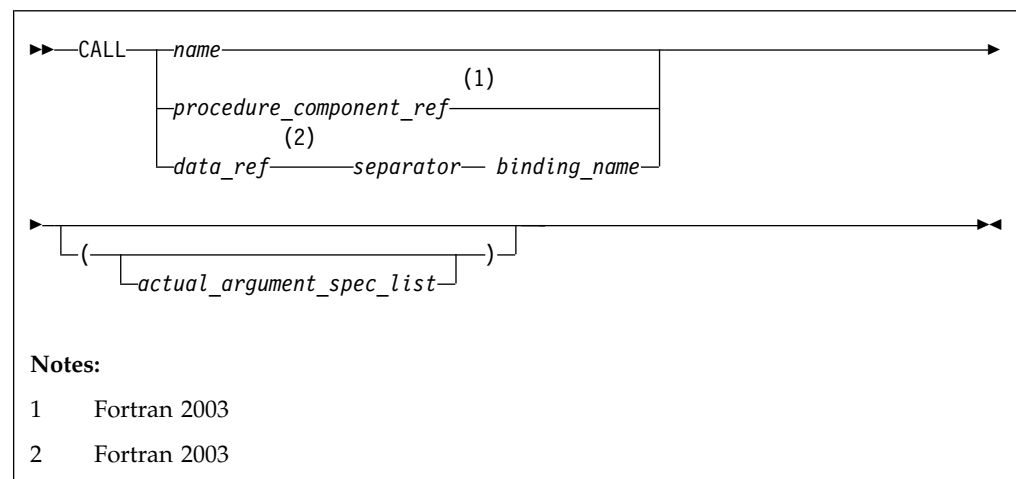- "Character" on page 44
- "Constant expressions" on page 102
- "Determining Type" on page 18 for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

CLASS (Fortran 2003)

Purpose

A **CLASS** type declaration statement specifies the declared type, type parameters, and attributes of objects of derived type. Initial values can be assigned to objects.
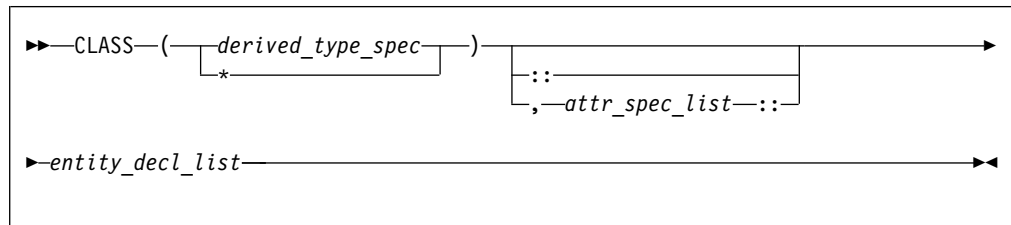
Syntax

```
>>--CLASS--(---+-derived_type_spec-+---)---+---------------------+-------------->
               '-*--------------'          +--::----------------+
                                           '-,--attr_spec_list--::-'

>--entity_decl_list------------------------------------><
```

derived_type_spec
> is the name of an extensible derived type. For more information, see "Type Declaration" on page 476.

attr_spec
> For more information, see "TYPE" on page 472.

entity_decl
> For more information, see "TYPE" on page 472.

Rules

The rules for the **TYPE** type declaration and **CLASS** type declaration are similar; for further information, see "TYPE" on page 472.

The following rules are unique to **CLASS** type declarations:

- The **CLASS** type specifier is used to declare a polymorphic object. The *type_name* is the declared type of a polymorphic object.
- Use the **CLASS(*)** specifier to declare an unlimited polymorphic object. An unlimited polymorphic entity is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.
- An entity declared with the **CLASS** keyword must be a dummy argument or have the **ALLOCATABLE** or **POINTER** attribute. Also, dummy arguments declared with the **CLASS** keyword must not have the value attribute.

Examples

```
program sClass

type base
  integer::i
end type

type,extends(base)::child
  integer::j
end type

type(child),target::child1=child(4,6)
type(base), target::base1=base(7)
! declare an item that could contain any extensible derived type
! or intrinsic type
```

```
          class(*),allocatable::anyThing

          ! declare basePtr as a polymorphic item with declared type base,
          ! could have run time type of base or child
          class(base),pointer::basePtr

          ! set basePtr to point to an item of type child
          basePtr=>child1
          call printAny(basePtr)

          ! set basePtr to point to an item of type base
          basePtr=>base1
          call printAny(basePtr)

          ! allocate an integer item
          allocate(anyThing, source=base1%i)
          call printAny(anyThing)

          contains

           subroutine printAny(printItem)
           ! declare a dummy arg of unlimited polymorphic, can point
           ! to any extensible derived type or intrinsic type
           class(*)::printItem

           select type(item=>printItem)
           type is (base)
            print*,' base item is ',item

           type is (child)
            print*,' child item is ', item

           type is (integer)
            print*,' integer item is ',item
           end select
          end subroutine
           end program
```

The output of the program is:

child item is 4 6
base item is 7
integer item is 7

Related information
• "Polymorphic entities (Fortran 2003)" on page 20

CLOSE

Purpose

The **CLOSE** statement disconnects an external file from a unit.

`F2003` Execution of a **CLOSE** statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit. `F2003`

Syntax

```
►►──CLOSE──(──close_list──)──────────────────────────────────►◄
```

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 expression. The integer expression has one of the following values:

- A value in the range 1 through 2147483647
- **F2008** A NEWUNIT value **F2008**

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

ERR= *stmt_label*

> is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

Fortran 2003

IOMSG= *iomsg_variable*

> is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

End of Fortran 2003

IOSTAT= *ios*

> is an input/output status specifier that specifies the status of the input/output operation. *ios* is an integer variable. When the input/output statement containing this specifier finishes executing, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

STATUS= *char_expr*

> specifies the status of the file after it is closed. *char_expr* is a scalar 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 default is **DELETE** if the file status is **SCRATCH**; otherwise, the default is **KEEP**.

Rules

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 scoping unit as the **OPEN** statement referring to that unit. You can specify a unit that does not exist or has no file connected; the **CLOSE** statement has no effect in this case.

IBM Unit 0 cannot be closed. **IBM**

When an executable program stops for reasons other than an error condition, all units that are connected are closed. Each unit is closed with the status **KEEP** unless the file status prior to completion was **SCRATCH**, in which case the unit is closed with the status **DELETE**. The effect is as though a **CLOSE** statement without a **STATUS=** specifier were executed on each connected unit.

If a preconnected unit is disconnected by a **CLOSE** statement, the rules of implicit opening apply if the unit is later specified in a **WRITE** statement (without having been explicitly opened).

Examples

```
CLOSE(15)
CLOSE(UNIT=16,STATUS='DELETE')
```

Related information
- "Units" on page 218
- "Conditions and IOSTAT values" on page 226
- "OPEN" on page 416

COMMON

Purpose

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

Syntax

object

```
 ►►─────variable_name──────────────────────────────────────────────────────►◄
          │           └(─explicit_shape_spec_list─)─┘│
          │                        (1)               │
          └─procedure_pointer_name───────────────────┘
```

Notes:

1   Fortran 2003
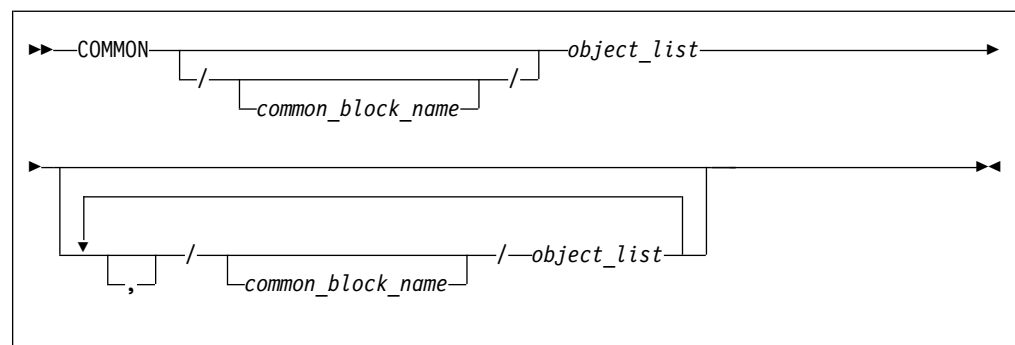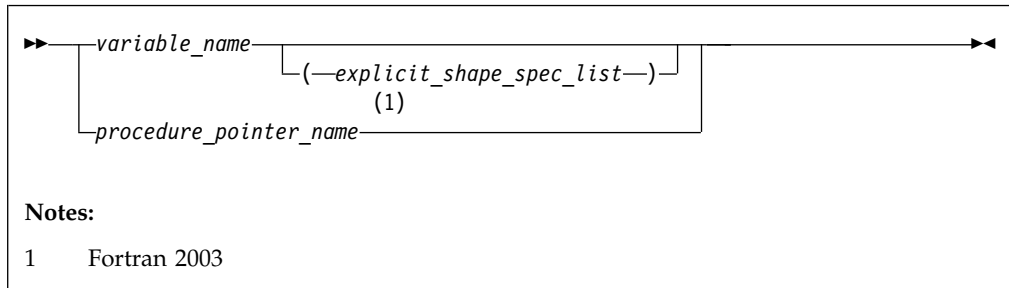
Rules

object cannot refer to a dummy argument, automatic object, allocatable object , or an object of a derived type that has an allocatable ultimate component, pointee, function, function result, or entry to a procedure, or a variable with the **BIND** attribute. *object* cannot have the **STATIC** or **AUTOMATIC** attributes.

If an *explicit_shape_spec_list* is present, *variable_name* must not have the **POINTER** attribute. Each dimension bound must be a constant specification expression. This form specifies that *variable_name* has the **DIMENSION** attribute.

`F2003` A given *variable_name* or procedure pointer name can only appear once in all common block object lists within a scoping unit. Their names cannot be made accessible by use association. `F2003`

If *object* is of derived type, it must be a sequence derived type. Given a sequenced structure where all the ultimate components are nonpointers, and are all of character type or all of type default integer, default real, default complex, default logical or double precision real, the structure is treated as if its components are enumerated directly in the common block.

Data pointers that are storage associated shall have deferred the same type parameters. Furthermore, a data pointer object in a common block can only be storage associated with pointers of the same type, type parameters, and rank.

An object in a common block with **TARGET** attribute can be storage associated with another object. That object must have the **TARGET** attribute and have the same type and type parameters.

`IBM` Pointers of type **BYTE** can be storage associated with pointers of type **INTEGER(1)** and **LOGICAL(1)**. Integer and logical pointers of the same length can be storage associated if you specify the **-qintlog** compiler option. `IBM`

`F2003` A procedure pointer can be storage associated only with another procedure pointer; both interfaces must be either explicit or implicit. If both interfaces are explicit, their characteristics must be the same. If both interfaces are implicit, both must be subroutines or both must be functions with the same type and type parameters. `F2003`

If you specify *common_block_name*, all variables specified in the *object_list* that follows are declared to be in that named common block. If you omit *common_block_name*, all variables that you specify in the *object_list* that follows are in the blank common block.

Within a scoping 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 are global entities.

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

See "Interoperability of common blocks" on page 888 for **BIND** information.

IBM extension

By default, common blocks are shared across threads, and so the use of the **COMMON** statement is thread-unsafe if any storage unit in the common block needs to be updated by more than one thread, or is updated by one thread and referenced by another. To ensure your application uses **COMMON** in a thread-safe manner, you must either serialize access to the data using locks, or make certain that the common blocks are local to each thread. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See Pthreads library module in the *XL Fortran Optimization and Programming Guide* for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See **CRITICAL /END CRITICAL** in the *XL Fortran Optimization and Programming Guide* for more information. The **THREADLOCAL** and **THREADPRIVATE** directives ensure that common blocks are local to each thread. See **THREADLOCAL** and **THREADPRIVATE** in the *XL Fortran Optimization and Programming Guide* for more information.

End of IBM extension

Related information:

Pthreads library module

Common association

Within an executable program, all nonzero-sized named common blocks with the same name have the same first storage unit. There can be one blank common block, and all scoping units that refer to nonzero-sized blank common refer to the same first storage unit.

All zero-sized common blocks with the same name are storage-associated with one another. All zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. Use association or host association can cause these associated objects to be accessible in the same scoping unit.

Because association is by storage unit, variables in a common block can have different names and types in different scoping units.

Common block storage sequence
Storage units for variables within a common block in a scoping unit are assigned 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, these statements specify X:

```
COMMON /X/ A,B      ! common block named X
REAL C(2)
EQUIVALENCE (B,C)
```

The contents of common block X are as follows:

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

Only **COMMON** and **EQUIVALENCE** statements that appear in a scoping unit contribute to the common block storage sequences formed in that unit, not including variables in common made accessible by use association or host association.

An **EQUIVALENCE** statement cannot cause the storage sequences of two different common blocks to become associated. While a common block can be declared in the scoping unit of a module `F2008` or submodule`F2008` , it must not be declared in another scoping unit that accesses entities from the module `F2008` or submodule`F2008` through use association.

Use of **COMMON** can lead to misaligned data. Any use of misaligned data can adversely affect 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 in the common block, including any extensions resulting from equivalence association.

Differences between named and blank common blocks

- Within an executable program, there can be more than one named common block, but only one blank common block.
- In all scoping 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 scoping units, the length of the longest block becomes the length of the blank common block in the executable program.)
- You can initially define objects in a named common block by using a **BLOCK DATA** program unit containing a **DATA** statement or a type declaration statement. You cannot initially define any elements of a blank common block.

  `IBM` If a named common block, or any part of it, is initialized in more than one scoping unit, the initial value is undefined. To avoid this problem, use block data program units or modules to initialize named common blocks; each named common block should be initialized in only one block data program unit or module. `IBM`

Examples

```
INTEGER MONTH,DAY,YEAR
COMMON /DATE/ MONTH,DAY,YEAR
REAL        R4
REAL        R8
CHARACTER(1)  C1
COMMON /NOALIGN/ R8,C1,R4    ! R4 will not be aligned on a
                             ! full-word boundary
```

Related information

- Pthreads library module in the *XL Fortran Optimization and Programming Guide*
- "BIND (Fortran 2003)" on page 298
- "Interoperability of common blocks" on page 888
- **THREADLOCAL** in the *XL Fortran Optimization and Programming Guide*.
- "Block data program unit" on page 190
- "Explicit-shape arrays" on page 77
- "Entities with scope" on page 151
- "Storage classes for variables (IBM extension)" on page 28

COMPLEX

Purpose

A **COMPLEX** type declaration statement specifies the length and attributes of objects and functions of type complex. Initial values can be assigned to objects.

Syntax

```
►►─COMPLEX──────────────────────────────────────entity_decl_list──►◄
            └─kind_selector─┘  ┌─::─────────────┐
                              └─,─attr_spec_list─::─┘
```

where:

attr_spec
       is any of the following:

ALLOCATABLE [1]	INTRINSIC	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE
AUTOMATIC [2]	PARAMETER	STATIC [2]
BIND [1]	POINTER	TARGET
DIMENSION (*array_spec*)	PRIVATE	VALUE [1]
EXTERNAL	PROTECTED [1]	VOLATILE
INTENT (*intent_spec*)		
Note: [1] Fortran 2003 [2] IBM extension		

kind_selector

```
       ┌──────────────────────int_constant_expr─)──┐
►►─┬─(─┤                                            ├─────────────────►◄
   │   └─KIND─ = ─┘                      (1)        │
   │                                                │
   └── * ─int_literal_constant──────────────────────┘
```

Notes:

1   IBM extension.

specifies the length of complex entities:
- If *int_constant_expr* is specified, the valid values are 4, 8 and 16. These values represent the precision and range of each part of the complex entity.
- If the **int_literal_constant* form is specified, the valid values are 8, 16 and 32. These values represent the length of the whole complex entity, and correspond to the values allowed for the alternative form. *int_literal_constant* cannot specify a kind type parameter.

attr_spec
For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
is either **IN**, **OUT**, or **INOUT**

::   is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
is a list of dimension bounds.

entity_decl

```
                            (1)          ┌─(─array_spec─)─┐
►►─a─┬───────────────────────────────────┴────────────────┴──────►
     │    └─ * ─len─┘                                       
     │                              (2)                     
     └─(─array_spec─)─ * ─len──────────────────────────────┘

►─┬──────────────────────────────────────────────────────►◄
  │                                   (3)                   
  ├─/─initial_value_list─/─────────────┘                    
  ├─ = ─constant_expr────────────                           
  └─ => ─NULL()──────────────                               
```

Notes:

1   IBM extension.

2   IBM extension.

3   IBM extension.

a   is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

> **IBM** *len*
>> overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications. **IBM**

> **IBM** *initial_value*
>> provides an initial value for the entity specified by the immediately preceding name. **IBM**

constant_expr
> provides a constant expression for the entity specified by the immediately preceding name.

=> **NULL()**
> provides an initial value for the pointer object

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if:
- it appears in a named common block in a block data program unit.
- **IBM** if it appears in a named common block in a module or submodule. **IBM**

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If *constant_expr* or **NULL()** is specified, and the entity you are declaring:
- is a variable, the variable is initially defined.
- is a derived type component, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit.

A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

`IBM` If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. `IBM`

Examples

```
COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
```

Related information
- "Complex" on page 41
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

CONTAINS

Purpose

The **CONTAINS** statement separates the body of a main program, external subprogram, or module subprogram from any internal subprograms that it may contain. Similarly, it separates the specification part of a module `F2008` or submodule `F2008` from any module subprograms.

F2003 The **CONTAINS** statement also introduces the procedure part of a derived-type definition. F2003

Syntax

```
►►──CONTAINS──────────────────────────────────────────────────►◄
```
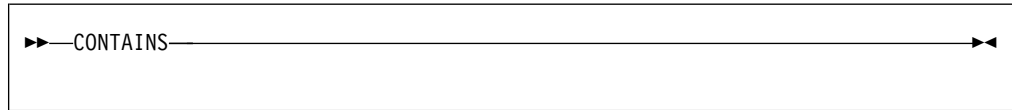
Rules

For a **CONTAINS** statement associated with subprograms, the following rules apply:

- The **CONTAINS** statement cannot appear in a block data program unit or in an internal subprogram.
- Any label of a **CONTAINS** statement is considered as part of the main program, subprogram, module, F2008 or submodule F2008 that contains the **CONTAINS** statement.

Examples

An example of a CONTAINS statement

```
MODULE A
   ...
  CONTAINS                  ! Module subprogram must follow
  SUBROUTINE B(X)
     ...
    CONTAINS                ! Internal subprogram must follow
    FUNCTION C(Y)
       ...
    END FUNCTION
  END SUBROUTINE
END MODULE
```

An example of a CONTAINS statement in a derived type definition

```
TYPE CUST
   INTEGER :: CUST_NUMBER
   CONTAINS
   PROCEDURE, PASS :: GET_CUST => GET_CUST_NUMBER
END TYPE CUST
```

Related information

- "Program units, procedures, and subprograms" on page 161

CONTIGUOUS (Fortran 2008)

Purpose

The **CONTIGUOUS** attribute specifies that the array elements of an array pointer, an assumed-shape array TS 29113 , or an assumed-rank object TS 29113 are not separated by other data objects.

An array pointer with the **CONTIGUOUS** attribute can only be pointer associated with a contiguous target. An assumed-shape array TS 29113 or an assumed-rank object TS 29113 with the **CONTIGUOUS** attribute is always contiguous; however,

the corresponding actual argument can be contiguous or noncontiguous. For details, see the "Rules" section.

Syntax

```
►►──CONTIGUOUS──┬────┬──contiguous_array_list────────────────►◄
                └─::─┘
```
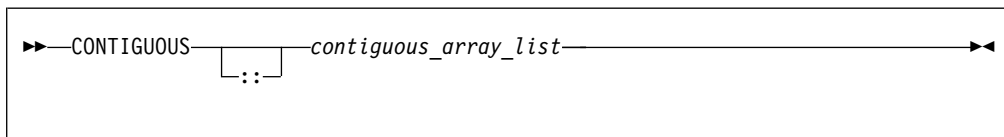
contiguous_array
>   an array that is contiguous

Rules

The entity that is specified with the **CONTIGUOUS** attribute must be an array pointer, an assumed-shape array TS 29113 , or an assumed-rank object TS 29113 .

In a pointer assignment, if the pointer has the **CONTIGUOUS** attribute, the target associated must be contiguous. The actual argument that corresponds to a pointer dummy argument with the **CONTIGUOUS** attribute must be simply contiguous.

If the actual argument that corresponds to an assumed-shape array dummy argument TS 29113 or an assumed-rank dummy argument TS 29113 with the **CONTIGUOUS** attribute is not contiguous, and the call is made from Fortran or the callee is a Fortran procedure, the compiler makes it contiguous by performing the following actions:

1. Create a temporary contiguous argument to associate with the dummy argument.
2. Initialize the temporary contiguous argument with the value of the actual argument.
3. When control returns from the procedure, copy the value of the temporary contiguous argument back to the actual argument.

   Note: The value is not copied back if the actual argument is specified as **INTENT(IN)**.

If an actual argument is a nonpointer array with the **ASYNCHRONOUS** or **VOLATILE** attribute but is not simply contiguous, and the corresponding dummy argument has either the **VOLATILE** or **ASYNCHRONOUS** attribute, that dummy argument must be an assumed-shape array without the **CONTIGUOUS** attribute.

If an actual argument is an array pointer with the **ASYNCHRONOUS** or **VOLATILE** attribute but without the **CONTIGUOUS** attribute, and the corresponding dummy argument has either the **VOLATILE** or **ASYNCHRONOUS** attribute, that dummy argument must be an array pointer or an assumed-shape array without the **CONTIGUOUS** attribute.

Compatible attributes

The following table lists the attributes that are compatible with the **CONTIGUOUS** attribute.

Table 39. Attributes compatible with the CONTIGUOUS attribute

AUTOMATIC **1**	OPTIONAL	SAVE
ASYNCHRONOUS	POINTER	STATIC **1**
DIMENSION	PRIVATE	TARGET
EXTERNAL	PROTECTED **2**	VOLATILE
INTENT	PUBLIC	

Notes:
   1 IBM extension
   2 Fortran 2003		

Examples

Example 1: CONTIGUOUS attribute specified for an array pointer

```
INTEGER, CONTIGUOUS, POINTER :: ap(:)
INTEGER, TARGET :: targ(10)
INTEGER, POINTER :: ip(:)
LOGICAL :: contig

! Invalid because ap is contiguous. A severe error is issued at compile time.
ap => targ(1:10:2)
ip => targ(1:10:2)
! contig has a value of .FALSE.
contig = IS_CONTIGUOUS(ip)

! contig has a value of .TRUE.
ALLOCATE(ip(10))
contig = IS_CONTIGUOUS(ip)
```

Example 2: CONTIGUOUS attribute specified for an assumed-shape array

```
LOGICAL :: contig

! Define a derived type named base
TYPE base(k, j, l)
  INTEGER, KIND :: k, j
  INTEGER, LEN :: l
  INTEGER(k) :: x
  INTEGER(j) :: y(l)
END TYPE

! Declare an allocatable, assumed-shape array b of base type
TYPE(base(4, 8, 0)), ALLOCATABLE :: b(:)
! Allocate two elements to b
ALLOCATE(b(2))
! contig has a value of .FALSE.
contig = IS_CONTIGUOUS(b%x)
```

Example 3: CONTIGUOUS attribute specified for an assumed-shape array

```
INTEGER, POINTER :: p(:)
INTEGER, TARGET :: t(10) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

p => t(1:10:2)

! The actual argument p, which corresponds to the contiguous dummy argument,
! is not contiguous. The compiler makes it contiguous by creating a temporary
! contiguous argument.
CALL fun(p)

CONTAINS
  SUBROUTINE fun(arg)
```

```
      ! Contiguous dummy argument arg
      INTEGER, CONTIGUOUS :: arg(:)
      PRINT *, arg(1)
    END SUBROUTINE
```

Related information
- IS_CONTIGUOUS
- "Contiguity (Fortran 2008)" on page 98
- "Argument association" on page 195

CONTINUE

Purpose

The **CONTINUE** statement is an executable control statement that takes no action; it has no effect. This statement is often used as the terminal statement of a loop.

Syntax

```
►►──CONTINUE────────────────────────────────────────────────────►◄
```
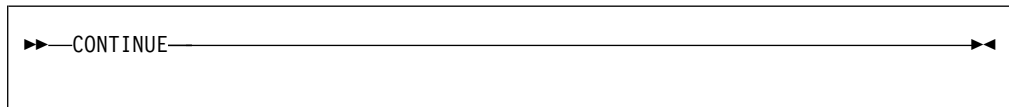
Examples

```
     DO 100 I = 1,N
        X = X + N
100  CONTINUE
```

Related information
- Chapter 7, "Execution control," on page 135

CYCLE

Purpose

The **CYCLE** statement terminates the current execution cycle of a **DO** or **DO WHILE** construct that it belongs to.

F2008 The **CYCLE** statement completes the execution of the current iteration of the **DO CONCURRENT** construct that it belongs to. F2008

Syntax

```
►►──CYCLE────────────────────────────────────────────────────►◄
          └─DO_construct_name─┘
```
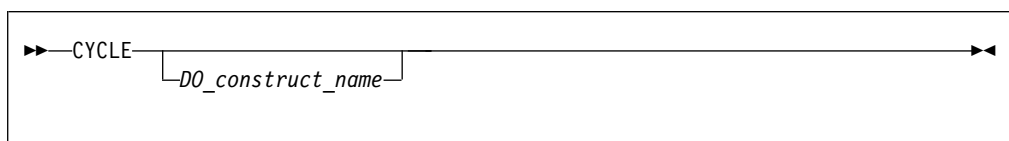
DO_construct_name
>   is the name of a **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** construct.

Rules

The **CYCLE** statement is placed within a **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct and belongs to the particular **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct that is specified by *DO_construct_name*. If *DO_construct_name* is not specified, the **CYCLE** statement belongs to the **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct that immediately surrounds it.

`F2008` A **CYCLE** statement must not be placed within a **DO CONCURRENT** construct if it belongs to an outer construct. `F2008`

Execution of a **CYCLE** statement that belongs to a **DO** or **DO WHILE** construct only terminates the current execution cycle of the **DO** or **DO WHILE** construct. Any executable statements after the **CYCLE** statement, including any terminating labeled action statement, will not be executed. For **DO** constructs, program execution continues with incrementation processing, if any. For **DO WHILE** constructs, program execution continues with loop control processing.

`F2008` Execution of a **CYCLE** statement that belongs to a **DO CONCURRENT** construct completes the execution of the current iteration of the construct. `F2008`

A **CYCLE** statement can have a statement label. However, it cannot be used as a labeled action statement that terminates a **DO** construct.

Examples

```
LOOP1: DO I = 1, 20
   N = N + 1
   IF (N > NMAX) CYCLE LOOP1        ! cycle to LOOP1

   LOOP2: DO WHILE (K==1)
      IF (K > KMAX) CYCLE           ! cycle to LOOP2
      K = K + 1
   END DO LOOP2

   LOOP3:  DO J = 1, 10
      N = N + 1
      IF (N > NMAX) CYCLE LOOP1     ! cycle to LOOP1
      CYCLE LOOP3                   ! cycle to LOOP3
   END DO LOOP3

END DO LOOP1
END
```

Related information

- "DO construct" on page 138
- "DO WHILE construct" on page 142
- "DO CONCURRENT construct (Fortran 2008)" on page 142

DATA

Purpose

The **DATA** statement provides initial values for variables.

Syntax

```
                   ┌──────┐
                   │  ┌─┐ │
                   ▼  │,│ │
  ►►──DATA──────data_object_list──/──initial_value_list──/──────────────────►◄
```
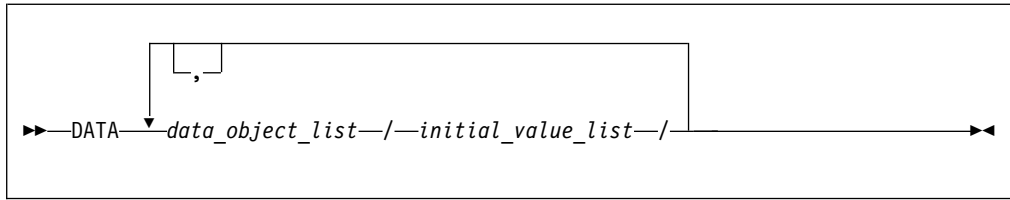
data_object

> is a variable or an implied-**DO** list. Any subscript or substring expression
> must be a constant expression.

*implied-**DO** list*

```
  ►►──(──do_object_list──,──do_variable── = ──integer_expr1──,──integer_expr2──────►
  ►────────────────────)──────────────────────────────────────────────────────►◄
      └,──integer_expr3─┘
```
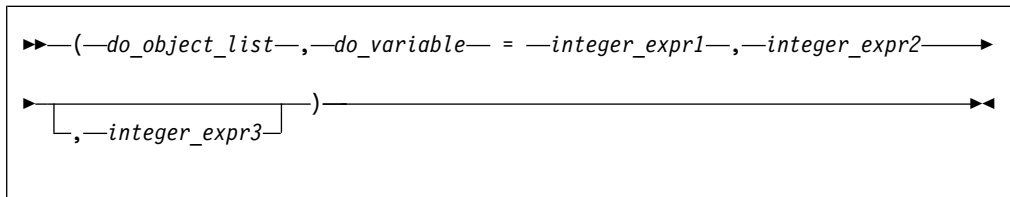
do_object

> is an array element, scalar structure component, substring, or
> implied-**DO** list

do_variable

> is a named scalar integer variable called the implied-**DO** variable.
> This variable is a statement entity.

integer_expr1, *integer_expr2*, **and** *integer_expr3*

> are each scalar integer expressions. The primaries of an expression
> can only contain constants or implied-**DO** variables of other
> implied-**DO** lists that have this implied-**DO** list within their
> ranges. Each operation must be intrinsic.

initial_value

```
  ►►──────────────data_value──────────────────────►◄
      └r── * ─┘
```
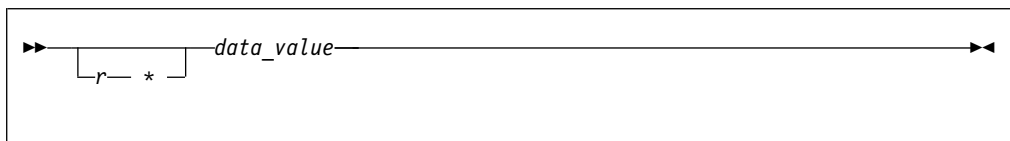
r	is a nonnegative scalar integer constant. If *r* is a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association.

> *r* may also be a nonnegative scalar integer subobject of a constant.
> Similar to the above paragraph, if it is a subobject of a named
> constant, it must have been declared previously in the scoping unit
> or made accessible by use or host association.

> If *r* is a subobject of a constant, any subscript in it is a constant
> expression. If *r* is omitted, the default value is 1. The form
> *r*data_value* is equivalent to *r* successive appearances of the data
> value.

data_value
> is a scalar constant, signed integer literal constant, signed real literal constant, structure constructor, scalar subobject of a constant, or **NULL()**.

Rules

If you specify that a nonpointer object has default initialization in a type definition, you cannot initialize that object with a **DATA** statement. Specifying a nonpointer array object as a *data_object* is the same as specifying a list of all the elements in the array object in the order they are stored.

An array with pointer attribute has only one corresponding initial value which is **NULL()**.

Each *data_object_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_object*.

For pointer initialization, if the *data_value* is **NULL()** then the corresponding *data_object* must have pointer attribute. If the *data_object* has pointer attribute then the corresponding *data_value* must be **NULL()**.

The definition of each *data_object* by its corresponding *initial_value* must follow the rules for intrinsic assignment, except as noted under "Using typeless constants" on page 33.

If *initial_value* is a structure constructor, each component must be a constant expression. If *data_object* is a variable, any substring, subscript, or stride expressions must be constant expressions.

If *data_value* is a named constant or a subobject of a named constant, the named constant must have been previously declared in the scoping unit, or made accessible by host or use association. If *data_value* is a structure constructor, the derived type must have been previously declared in the scoping unit, or made accessible by host or use association.

Zero-sized arrays, implied-**DO** lists with iteration counts of zero, and values with a repeat factor of zero contribute no variables to the expanded *initial_value_list*, although a zero-length scalar character variable contributes one variable to the list.

You can use an implied-**DO** list in a **DATA** statement to initialize array elements, scalar structure components and substrings. The implied-**DO** list is expanded into a sequence of scalar structure components, array elements, or substrings, under the control of the implied-**DO** variable. Array elements and scalar structure components must not have constant parents. Each scalar structure component must contain at least one component reference that specifies a subscript list.

The range of an implied-**DO** list is the *do_object_list*. The iteration count and the values of the implied-**DO** variable are established from *integer_expr1*, *integer_expr2*, and *integer_expr3*, the same as for a **DO** statement. When the implied-**DO** list is executed, 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** variables. If the implied-**DO** variable has an iteration count of 0, no variables are added to the expanded sequence.

Each subscript expression in a *do_object* can only contain constants or implied-**DO** variables of implied-**DO** lists that have the subscript expression within their ranges. Each operation must be intrinsic.

IBM extension

To initialize list items of type logical with logical constants, you can also use the abbreviated forms (T for .TRUE. and F for .FALSE.). If T or F is a constant name that was defined previously with the **PARAMETER** attribute, XL Fortran recognizes it as the named constant and assigns its value to the corresponding list item in the **DATA** statement.

End of IBM extension

In a block data program unit, you can use a **DATA** statement or type declaration statement to provide an initial value for a variable in a named common block.

In an internal or module subprogram, if the *data_object* is the same name as an entity in the host, and the *data_object* is not declared in any other specification statement in the internal subprogram, the *data_object* must not be referenced or defined before the **DATA** statement.

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

- An automatic object.
- A dummy argument.
- **IBM** A pointee. **IBM**
- A variable in a blank common block.
- The result variable of a function.
- **IBM** A data object whose storage class is automatic. **IBM**
- **F2003** A variable that has the **ALLOCATABLE** attribute. **F2003**

You must not initialize a variable more than once in an executable program. If you associate two or more variables, you can only initialize one of the data objects.

Examples

Example 1:

```
      INTEGER Z(100),EVEN_ODD(0:9)
      LOGICAL FIRST_TIME
      CHARACTER*10 CHARARR(1)
      DATA    FIRST_TIME / .TRUE. /
      DATA    Z  / 100* 0 /
! Implied-DO list
      DATA  (EVEN_ODD(J),J=0,8,2) / 5 * 0 /  &
   &        ,(EVEN_ODD(J),J=1,9,2) / 5 * 1 /
! Nested example
      DIMENSION TDARR(3,4)  ! Initializes a two-dimensional array
      DATA ((TDARR(I,J),J=1,4),I=1,3) /12 * 0/
! Character substring example
      DATA (CHARARR(J)(1:3),J=1,1) /'aaa'/
      DATA (CHARARR(J)(4:7),J=1,1) /'bbbb'/
      DATA (CHARARR(J)(8:10),J=1,1) /'ccc'/
! CHARARR(1) contains 'aaabbbbccc'
```

Example 2:

```
TYPE DT
     INTEGER :: COUNT(2)
END TYPE DT

TYPE(DT), PARAMETER, DIMENSION(3) :: SPARM = DT ( (/3,5/) )

INTEGER :: A(5)

DATA A /SPARM(2)%COUNT(2) * 10/
```

Related information
- "Executing a DO statement" on page 140
- "Statement and construct entity" on page 155

DEALLOCATE

Purpose

The **DEALLOCATE** statement dynamically deallocates allocatable objects and pointer targets. A specified pointer becomes disassociated, while any other pointers associated with the target become undefined.

Syntax

```
►►──DEALLOCATE──────────────────────────────────────────────────────►

►─(──allocate_object_list──────────────────────────────────────)───►◄
                        └─,──STAT── = ──stat_variable─┘
                                              └─,──ERRMSG── = ──errmsg_variable──┘ (1)

Notes:

1      Fortran 2003
```
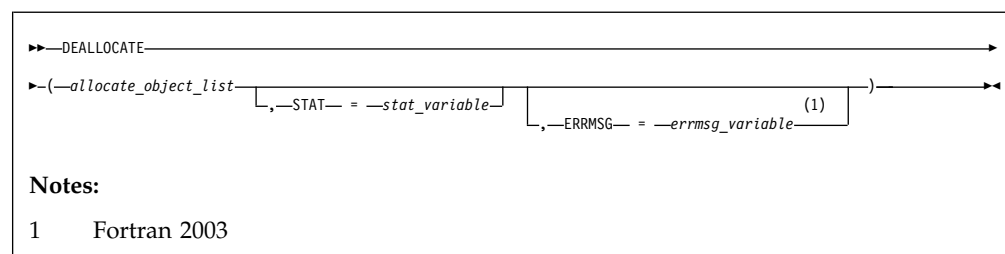
allocate_object
  is a data pointer or an allocatable object

stat_variable
  is a scalar integer variable

F2003 *errmsg_variable*
  is a scalar character variable **F2003**

Rules

An allocatable object that appears in a **DEALLOCATE** statement must be allocated.

F2003 When the result of a referenced function is allocatable, or has a structure with allocatable subobjects, that result and any allocated allocatable subobjects are deallocated after execution of the innermost executable construct containing the function reference. **F2003**

An allocatable object with the **TARGET** attribute cannot be deallocated through an associated pointer. Deallocation of such an object causes the association status of any associated pointer to become undefined. An allocatable object that has an undefined allocation status cannot be subsequently referenced, defined, allocated,

or deallocated. Successful execution of a **DEALLOCATE** statement causes the allocation status of an allocatable object to become not allocated.

`F2003` An object being deallocated will be finalized first. When a variable of derived type is deallocated, any allocated subobject with the **ALLOCATABLE** attribute is also deallocated. If an allocatable component is a subobject of a finalizable object, that object is finalized before the component is automatically deallocated. `F2003`

When an intrinsic assignment statement is executed, any allocated subobject of the variable is deallocated before the assignment takes place.

A pointer that appears in a **DEALLOCATE** statement must be associated with a whole target that was created with an **ALLOCATE** statement. Deallocation of a pointer target causes the association status of any other pointer associated with all or part of the target to become undefined.

Deallocation of a variable containing allocatable components automatically deallocates all such components of the variable that are allocated.

Tips

Use the **DEALLOCATE** statement instead of the **NULLIFY** statement if no other pointer is associated with the allocated memory.

Deallocate memory that a pointer function has allocated.

If the **STAT=** specifier is not present and an error condition occurs during execution of the statement, the program terminates. If the **STAT=** specifier is present, *stat_variable* is assigned one of the following values:

IBM extension

Stat value	Error condition
0	No error
1	Error in system routine attempting to do deallocation
2	An invalid data object has been specified for deallocation
3	Both error conditions 1 and 2 have occurred

End of IBM extension

`F2003` If an error condition occurs during execution of the **DEALLOCATE** statement, an explanatory message is assigned to *errmsg_variable*. If no such condition occurs, the value of *errmsg_variable* is not changed. `F2003`

An *allocate_object* must not depend on the value, bounds, allocation status, or association status of another *allocate_object* in the same **DEALLOCATE** statement; nor does it depend on the value of the *stat_variable* `F2003` or *errmsg_variable* `F2003` in the same **DEALLOCATE** statement.

stat_variable and *errmsg_variable* must not be deallocated within the same
DEALLOCATE statement. The variable must not depend on the value, bounds,
allocation status, or association status of any *allocate_object* in the same
DEALLOCATE statement.

Examples

```
INTEGER, ALLOCATABLE :: A(:,:)
INTEGER X,Y
      .
      .
      .
ALLOCATE (A(X,Y))
      .
      .
      .
DEALLOCATE (A,STAT=I)
END
```

Related information

- "ALLOCATE" on page 287
- "ALLOCATABLE (Fortran 2003)" on page 285
- "Allocation status" on page 28
- "Pointer association" on page 159
- "Deferred-shape arrays" on page 80
- "Allocatable objects as dummy arguments (Fortran 2003)" on page 203

TYPE

Purpose

The **TYPE** statement is the first statement of a derived type definition.

Syntax

```
>>--TYPE--+----------------------+--type_name--+-----------------------------+--><
          +--::------------------+             +-(--type_param_name_list--)--+
          +-,type_attribute_list ::-+
```
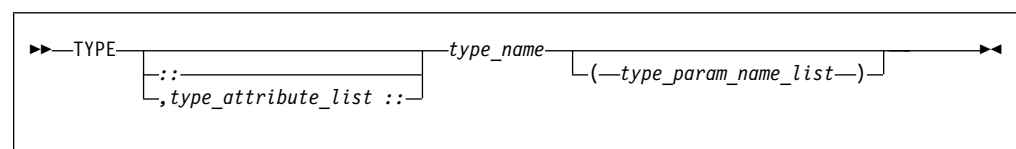
type_attribute
>   is **PRIVATE**, **PUBLIC**, `F2003` **BIND(C)**, *extends_spec*, or **ABSTRACT**
>   `F2003`.

type_name
>   is the name of the derived type

extends_spec **(Fortran 2003)**
>   is

```
>>--EXTENDS--(--parent_type_name--)------------------------------------------><
```
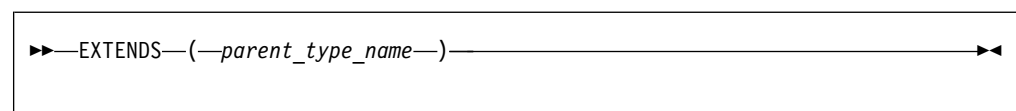
type_param_name **(Fortran 2003)**

is the name of a type parameter. For more information, see "Derived type parameters (Fortran 2003)" on page 50.

```
►►──┬─type_param_name─┬──────────────────────────────────────►◄
    └─,───────────────┘
```
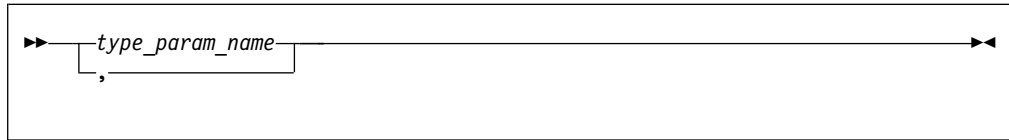
Rules

`F2003` The same type attribute can not occur more than once on the same TYPE statement. `F2003`

You can specify the **PRIVATE** or **PUBLIC** attribute only if the derived type definition is within the specification part of a module. A derived type definition can be **PRIVATE** or **PUBLIC**, not both.

If you define a type as **PRIVATE**, the following are accessible only within the defining module:
- The type name and any `F2003` type parameter names `F2003` for this derived type.
- Structure constructors for the type.
- Any procedure that has a dummy argument or function result of the type.

The *type_name* must not be the same as the name of any intrinsic type, except **BYTE** and **DOUBLECOMPLEX**. The *type_name* must also not be the name of any other accessible derived type.

─────────────────────────── **Fortran 2003** ───────────────────────────

BIND(C) explicitly defines the Fortran derived type as interoperable with a C type. The components must be of interoperable types. (See "Interoperability of types" on page 885 for additional information.) A derived type with the **BIND** attribute cannot be a **SEQUENCE** type. A component of a derived type with the **BIND** attribute must have interoperable type and type parameters, and cannot have the **POINTER** or **ALLOCATABLE** attribute.

A derived type with the **BIND** attribute cannot have type parameters.

The *parent_type_name* must be an accessible extensible type.

You can only specify the **ABSTRACT** attribute for an extensible type.

If **EXTENDS** is specified, **SEQUENCE** cannot appear for that type.

If **EXTENDS** is specified and the type being defined has an ultimate component of type **LOCK TYPE** from the **ISO_FORTRAN_ENV** intrinsic module, its parent type must have an ultimate component of type **LOCK TYPE**.

If **EXTENDS** is specified, the type must not have the **BIND(C)** attribute.

─────────────────────────── **End of Fortran 2003** ───────────────────────────

If the corresponding **END TYPE** statement specifies a name, it must be the same as *type_name*.

Examples

```
MODULE ABC
  TYPE, PRIVATE :: SYSTEM       ! Derived type SYSTEM can only be accessed
    SEQUENCE                    !   within module ABC
    REAL :: PRIMARY
    REAL :: SECONDARY
    CHARACTER(20), DIMENSION(5) :: STAFF
  END TYPE
END MODULE
 TYPE MULTIDIM (K,NDIMS)
    INTEGER, KIND :: K
    INTEGER, LEN  :: NDIMS
    REAL(K) :: POS(NDIMS)
 END TYPE MULTIDIM
 TYPE, EXTENDS(MULTIDIM) :: NAMED_MULTI (L)
    INTEGER, LEN :: L
    CHARACTER(L) :: NAME
 END TYPE NAMED_MULTI
```

Related information

- Chapter 4, "Derived types," on page 49
- "Interoperability of types" on page 885
- "END TYPE" on page 355
- "SEQUENCE" on page 460
- "Extensible derived types (Fortran 2003)" on page 58
- "Abstract types and deferred bindings (Fortran 2003)" on page 60

DIMENSION

Purpose

The **DIMENSION** attribute specifies that an entity is an array.

Syntax
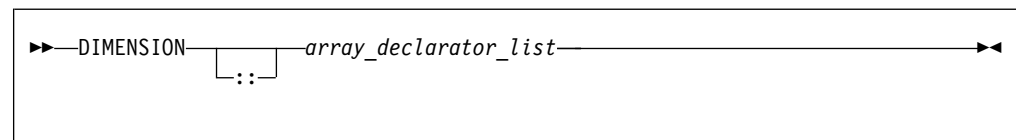
```
►►──DIMENSION──┬────┬──array_declarator_list────────────────────────►◄
               └─::─┘
```

array_declarator_list
       is a list of array declarators that specifies the rank or rank and shape of an array.

Rules

The Fortran standard allows arrays with up to fifteen dimensions.

IBM With XL Fortran, you can specify up to twenty dimensions. **IBM**

Only one dimension specification for an array name can appear in a scoping unit.

Table 40. Attributes compatible with the DIMENSION attribute

ALLOCATABLE **1**	OPTIONAL	PUBLIC
ASYNCHRONOUS	PARAMETER	SAVE
AUTOMATIC **3**	POINTER	STATIC **3**
BIND **1**	PRIVATE	TARGET
CONTIGUOUS **2**	PROTECTED **1**	VOLATILE
INTENT		
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

```
CALL SUB(5,6)
CONTAINS
SUBROUTINE SUB(I,M)
  DIMENSION LIST1(I,M)                        ! automatic array
  INTEGER, ALLOCATABLE, DIMENSION(:,:) :: A   ! deferred-shape array

:
END SUBROUTINE
END
```

Related information
- Chapter 5, "Array concepts," on page 75
- "VIRTUAL (IBM extension)" on page 489

DO

Purpose

The **DO** statement controls the execution of the statements that follow it, up to and including a specified terminal statement. Together, these statements form a **DO** construct.

Syntax
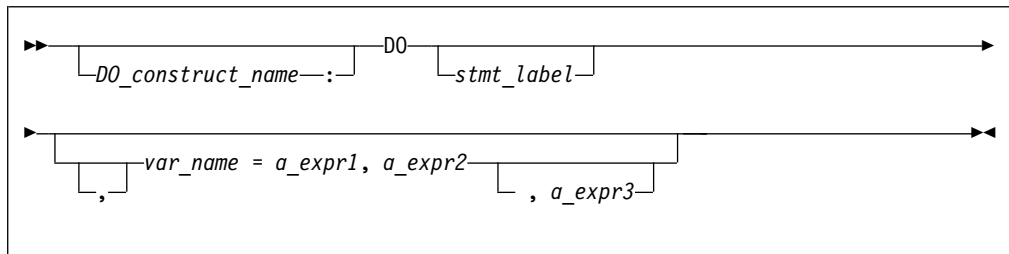
DO_construct_name
　　　　　is a name that identifies the **DO** construct.

stmt_label
　　　　　is the statement label of an executable statement appearing after the **DO** statement in the same scoping unit. This statement denotes the end of the **DO** construct.

var_name
        is a scalar variable name of type integer or real, called the **DO** variable

a_expr1, *a_expr2*, **and** *a_expr3*
        are each scalar expressions of type integer or real

Rules

If you specify a *DO_construct_name* on the **DO** statement, you must terminate the construct with an **END DO** and the same *DO_construct_name*. Conversely, if you do not specify a *DO_construct_name* on the **DO** statement, and you terminate the **DO** construct with an **END DO** statement, you must not have a *DO_construct_name* on the **END DO** statement.

If you specify a statement label in the **DO** statement, you must terminate the **DO** construct with a statement that is labeled with that statement label. You can terminate a labeled **DO** 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** statement, you must terminate the **DO** construct with an **END DO** statement.

If the control clause (the clause beginning with *var_name*) is absent, the statement is an infinite **DO**. The loop will iterate indefinitely until interrupted (for example, by the **EXIT** statement).

Examples

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO M=1,10
    READ (5,*) J
    IF (J.LE.I) THEN
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
    IF (SUM.GT.500) EXIT OUTER
    IF (SUM.GT.100) EXIT INNER
  END DO INNER
  SUM=SUM+I
  I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

Related information

- "DO construct" on page 138
- "END (Construct)" on page 351, for details on the **END DO** statement
- "EXIT" on page 366
- "CYCLE" on page 326
- "INDEPENDENT" on page 521
- "ASSERT" on page 508
- "CNCALL" on page 512
- "PERMUTATION" on page 531
- **PARALLEL DO/END PARALLEL DO** in the *XL Fortran Optimization and Programming Guide*

DO CONCURRENT (Fortran 2008)

Purpose

The **DO CONCURRENT** statement is the first statement in the **DO CONCURRENT** construct, which allows you to specify that individual loop iterations have no interdependencies.

Syntax

where *concurrent_header* is:

where *concurrent_control* is:
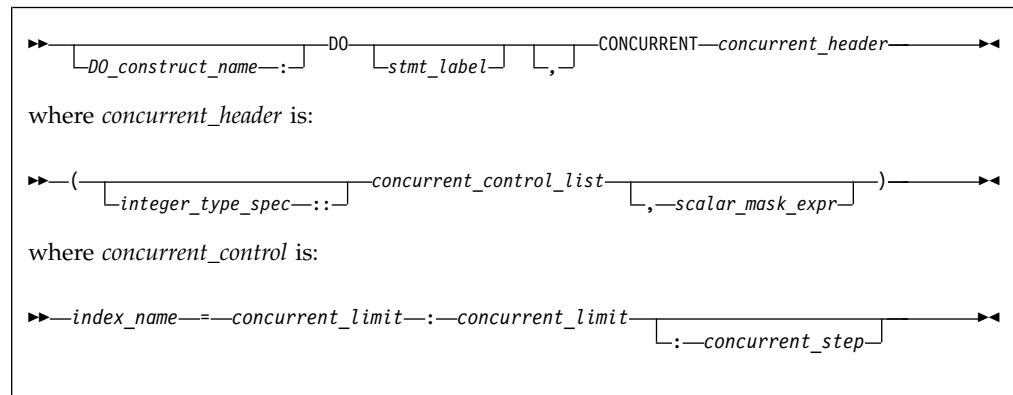
DO_construct_name
> is a name that identifies the **DO CONCURRENT** construct.

stmt_label
> is the statement label of an executable statement that appears after the **DO CONCURRENT** statement in the same scoping unit. It denotes the end of the **DO CONCURRENT** construct.

scalar_mask_expr
> is a scalar logical expression. It is evaluated for each combination of *index-name* values the same way as if it were used in the **FORALL** construct.

index_name
> is a named variable that is a scalar integer. It has a scope of the **DO CONCURRENT** construct.

concurrent_limit, *concurrent_step*
> are both scalar integer expressions. They can be evaluated in any order that is valid in the **FORALL** construct.

Constraints

- *concurrent_limit* or *concurrent_step* in a *concurrent_control* must not contain a reference to any *index_name* in the *concurrent_control_list* in which the *concurrent_limit* or *concurrent_step* parameter appears.
- A **CYCLE** statement must not appear within a **DO CONCURRENT** construct if it belongs to an outer construct.
- A **RETURN** statement must not appear within a **DO CONCURRENT** construct.
- A branch within a **DO CONCURRENT** construct must not have a branch target that is outside the construct.
- A reference to a nonpure procedure must not appear within a **DO CONCURRENT** construct.

- A reference to the procedure **IEEE_GET_FLAG**, **IEEE_SET_HALTING_MODE**, or **IEEE_GET_HALTING_MODE** from the intrinsic module **IEEE_EXCEPTIONS** must not appear within a **DO CONCURRENT** construct.
- An **EXIT** statement must not appear within a **CRITICAL** or **DO CONCURRENT** construct if it belongs to that construct or an outer construct.

Rules

If the **DO** statement of a **DO** construct specifies a *DO_construct_name*, the corresponding *END_DO_statement* must specify the same *DO_construct_name*; otherwise, the corresponding *END_DO_statement* must not specify a *DO_construct_name*.

Execution of a **CYCLE** statement that belongs to a **DO CONCURRENT** construct completes execution of that iteration of the construct.

If *integer_type_spec* appears in *concurrent_header*, it has the specified type and type parameters; otherwise, the following rules apply:
- Its type and type parameters are the same as the type and type parameters it would have if it were the name of a variable in the innermost executable construct or scoping unit. In addition, this type must be integer.
- An *index_name* must not be the same as an accessible global identifier, local identifier, or identifier of an outer construct entity, except for a common block name or a scalar variable name.

The set of combinations of *index_name* values is the Cartesian product of the sets that is defined by each control specification. The set of combinations of *index_name* values is active when *scalar_mask_expr* has the value .TRUE.. An *index_name* becomes defined when this set is evaluated. The range of a **DO CONCURRENT** construct is executed for every active combination of the *index_name* values. Each execution of the range is an iteration. The executions might occur in any order.

An *index_name* of a contained **DO CONCURRENT** construct must not be the same as an *index_name* of any of its containing **DO CONCURRENT** constructs.

For a **DO CONCURRENT** construct, the loop terminates and the **DO** construct becomes inactive when execution of all the iterations is completed.

When a **DO CONCURRENT** construct terminates, the following rules apply:
- The association status of a pointer is processor dependent if the pointer's association status was changed in more than one iteration of the construct.
- A variable that is defined or becomes undefined during more than one iteration of the construct becomes undefined.

The ordering among records that are written by different iterations of a **DO CONCURRENT** construct is processor dependent if the records are written to a file that is connected for sequential access by more than one iteration.

The following rules are related to execution of a **DO CONCURRENT** construct:
- A variable that is referenced in an iteration must either be previously defined during that iteration, or must not be defined or become undefined during any other iteration. A variable that is defined or becomes undefined by more than one iteration becomes undefined when the loop terminates. This rule applies to any procedure that is invoked within the loop.

- If a pointer is used in an iteration other than in the case of pointer assignment, allocation, or nullification, it must either be previously pointer associated during that iteration or must not have its pointer association changed during any other iteration. A pointer that has its pointer association changed in more than one iteration has an undefined association status when the construct terminates.
- If an allocatable object is referenced, defined, deallocated, or has its allocation status, dynamic type, or a deferred type parameter value inquired about in an iteration, it must either be previously allocated in that iteration or must not be allocated or deallocated in any other iteration. If an allocatable object is allocated in more than one iteration, it must have an allocation status of unallocated at the end of every iteration.
- If data are written to a file record or position in one iteration, that record or position in that file must not be read from or written to in a different iteration.

Related information

- "CYCLE" on page 326
- "DO CONCURRENT construct (Fortran 2008)" on page 142
- "END (Construct)" on page 351
- "EXIT" on page 366
- "RETURN" on page 454

DO WHILE

Purpose

The **DO WHILE** statement is the first statement in the **DO WHILE** construct, which indicates that you want the following statement block, 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.

Syntax

```
>>─────────────────────────────DO───────────────────────────────────────>
        └─DO_construct_name─:─┘    └─stmt_label─┘  └─,─┘

>─WHILE──(──logical_expr──)──────────────────────────────────────────><
```
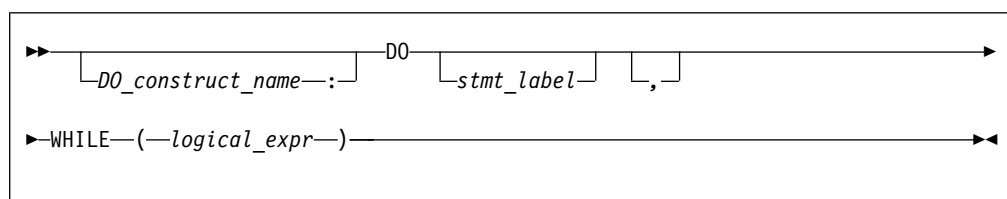
DO_construct_name
      is a name that identifies the **DO WHILE** construct

stmt_label
      is the statement label of an executable statement appearing after the **DO WHILE** statement in the same scoping unit. It denotes the end of the **DO WHILE** construct.

logical_expr
      is a scalar logical expression

Rules

If you specify a *DO_construct_name* on the **DO WHILE** statement, you must terminate the construct with an **END DO** and the same *DO_construct_name*.

Conversely, if you do not specify a *DO_construct_name* on the **DO WHILE** statement, and you terminate the **DO WHILE** construct with an **END DO** statement, you must not have a *DO_construct_name* on the **END DO** statement.

If you specify a statement label in the **DO WHILE** statement, you must terminate the **DO WHILE** construct 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** construct with an **END DO** statement.

Examples

```
      MYDO: DO 10 WHILE (I .LE. 5)  ! MYDO is the construct name
         SUM = SUM + INC
         I = I + 1
10    END DO MYDO
      END

      SUBROUTINE EXAMPLE2
        REAL X(10)
        LOGICAL FLAG1
        DATA    FLAG1 /.TRUE./
        DO 20 WHILE (I .LE. 10)
           X(I) = A
           I = I + 1
20      IF (.NOT. FLAG1) STOP
      END SUBROUTINE EXAMPLE2
```

Related information

- "DO WHILE construct" on page 142
- "END (Construct)" on page 351, for details on the **END DO** statement
- "EXIT" on page 366
- "CYCLE" on page 326

DOUBLE COMPLEX (IBM extension)

Purpose

A **DOUBLE COMPLEX** type declaration statement specifies the attributes of objects and functions of type double complex. Initial values can be assigned to objects.

Syntax

```
►►──DOUBLE COMPLEX──┬──────────────────────────┬──entity_decl_list──────►◄
                    ├──::─────────────────────┤
                    └──,──attr_spec_list──::──┘
```

where:

attr_spec
        is any of the following:

ALLOCATABLE **1**	INTRINSIC	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE
AUTOMATIC **2**	PARAMETER	STATIC **2**
BIND **1**	POINTER	TARGET
DIMENSION (*array_spec*)	PRIVATE	VALUE **1**
EXTERNAL	PROTECTED **1**	VOLATILE
INTENT (*intent_spec*)		
Note: **1** Fortran 2003 **2** IBM extension		

attr_spec
> For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**

array_spec
> is a list of dimension bounds

entity_decl

a is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

initial_value
> provides an initial value for the entity specified by the immediately preceding name

constant_expr
> provides a constant expression for the entity specified by the immediately preceding name

=> **NULL()**
> provides the initial value for the pointer object

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.

- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module `F2008` or submodule `F2008`.

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant.

Examples

```
SUBROUTINE SUB
  DOUBLE COMPLEX, STATIC, DIMENSION(1) :: B
END SUBROUTINE
```

Related information

- "COMPLEX" on page 319
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

DOUBLE PRECISION

Purpose

A **DOUBLE PRECISION** type declaration statement specifies the attributes of objects and functions of type double precision. Initial values can be assigned to objects.

Syntax

```
►►──DOUBLE PRECISION──┬─────────────────────┬──entity_decl_list──────────►◄
                      ├──::─────────────────┤
                      └──,──attr_spec_list──::──┘
```

where:

attr_spec
    is any of the following:

ALLOCATABLE [1]	**INTRINSIC**	**PUBLIC**
ASYNCHRONOUS	**OPTIONAL**	**SAVE**
AUTOMATIC [2]	**PARAMETER**	**STATIC** [2]
BIND [1]	**POINTER**	**TARGET**
DIMENSION (*array_spec*)	**PRIVATE**	**VALUE** [1]
EXTERNAL	**PROTECTED** [1]	**VOLATILE**
INTENT (*intent_spec*)		
Note: [1] Fortran 2003 [2] IBM extension		

attr_spec
> For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

::
> is the double colon separator. Use the double colon separator when you specify attributes, =*initialization_expr*, or => **NULL()**

array_spec
> is a list of dimension bounds
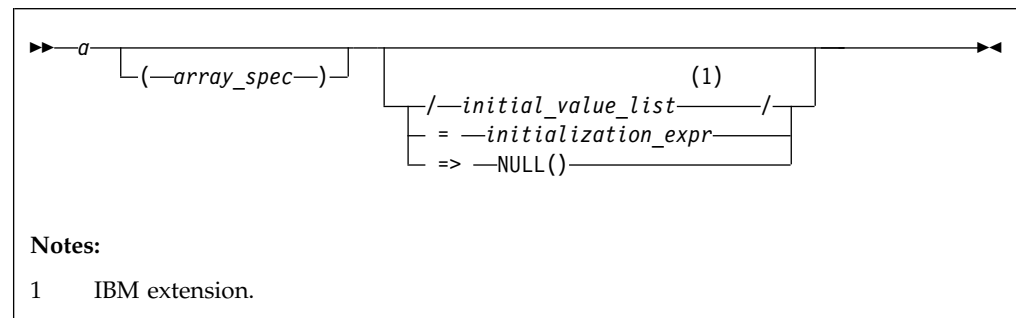
entity_decl

```
►►─┬─a─────────────────────────────────────────────────────────────┬─►◄
   │  └─(─array_spec─)─┘                                  (1)        │
   │                     ┌─/─initial_value_list─────────/─┐          │
   │                     ├─ = ─initialization_expr────────┤          │
   │                     └─ => ─NULL()──────────────────────┘          │
```

Notes:

1   IBM extension.

---

  a
>   is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

 IBM *initial_value*
>   provides an initial value for the entity specified by the immediately preceding name. **IBM**

 initialization_expr
>   provides an initial value, by means of a constant expression, for the entity specified by the immediately preceding name.

 => **NULL()**
>   provides the initial value for the pointer object.

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *initialization_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *initialization_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit **IBM** or if it appears in a named common block in a module or submodule. **IBM**

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

initialization_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *initialization_expr* or **NULL()** is specified, the variable is initially defined. If the entity you are declaring is a derived type component, and *initialization_expr* or **NULL()** is specified, the derived type has default initialization. *a* becomes defined with the value determined by *initialization_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *initialization_expr* or => **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

IBM If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. **IBM**

Examples

```
DOUBLE PRECISION, POINTER :: PTR
DOUBLE PRECISION, TARGET  :: TAR
```

Related information

- "REAL" on page 449
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

ELSE

Purpose

The **ELSE** statement is the first statement of the optional **ELSE** block within an **IF** construct.

Syntax

```
►►──ELSE──────────────────────────────────────────────►◄
           └─IF_construct_name─┘
```
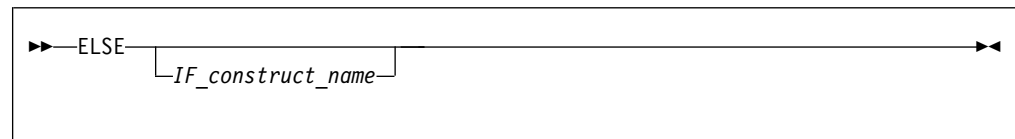
IF_construct_name
        is a name that identifies the **IF** construct

Syntax

Control branches to the **ELSE** block if every previous logical expression in the **IF** construct evaluates as false. The statement block of the **ELSE** block is executed and the **IF** construct is complete.

If you specify an *IF_construct_name*, it must be the same name that you specified in the block **IF** statement.

Examples

```
IF (A.GT.0) THEN
  B = B-A
ELSE              ! the next statement is executed if a<=0
  B = B+A
END IF
```

Related information

- "IF construct" on page 143
- "END (Construct)" on page 351, for details on the **END IF** statement
- "ELSE IF" on page 348

ELSE IF

Purpose

The **ELSE IF** statement is the first statement of an optional **ELSE IF** block within an **IF** construct.

Syntax

```
►►──ELSE IF──(──scalar_logical_expr──)──THEN─────────────────────────►◄
                                          └─IF_construct_name─┘
```
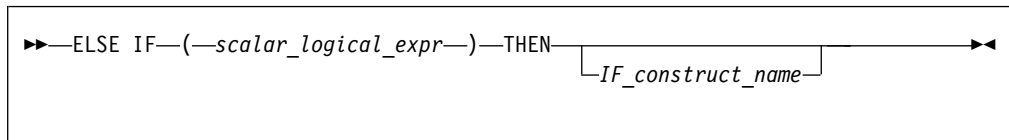
IF_construct_name
> is a name that identifies the **IF** construct

Rules

scalar_logical_expr is evaluated if no previous logical expressions in the **IF** construct are evaluated as true. If *scalar_logical_expr* is true, the statement block that follows is executed and the **IF** construct is complete.

If you specify an *IF_construct_name*, it must be the same name that you specified in the block **IF** statement.

Examples

```
IF (I.EQ.1) THEN
    J=J-1
ELSE IF (I.EQ.2) THEN
    J=J-2
ELSE IF (I.EQ.3) THEN
    J=J-3
ELSE
    J=J-4
END IF
```

Related information
- "IF construct" on page 143
- "END (Construct)" on page 351, for details on the **END IF** statement
- "ELSE" on page 347

ELSEWHERE

Purpose

The **ELSEWHERE** statement is the first statement of the optional **ELSEWHERE** or masked **ELSEWHERE** block within a **WHERE** construct.
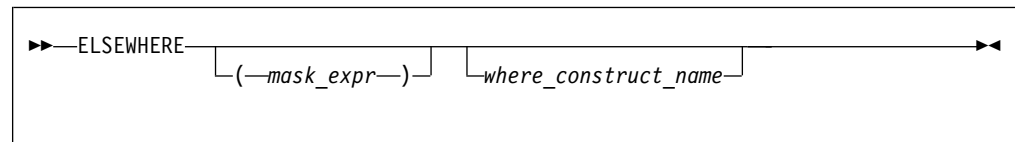
Syntax

```
>>--ELSEWHERE----------------------------------------------------><
             |_(--mask_expr--)_|  |_where_construct_name_|
```

mask_expr
    is a logical array expression

where_construct_name
    is a name that identifies a **WHERE** construct

Rules

A masked **ELSEWHERE** statement contains a *mask_expr*. See "Interpreting masked array assignments" on page 123 for information on interpreting mask expressions. Each *mask_expr* in a **WHERE** construct must have the same shape.

If you specify a *where_construct_name*, it must be the same name that you specified on the **WHERE** construct statement.

ELSEWHERE and masked **ELSEWHERE** statements must not be branch target statements.

Examples

The following example shows a program that uses a simple masked **ELSEWHERE** statement to change the data in an array:

```
INTEGER ARR1(3, 3), ARR2(3,3), FLAG(3, 3)

ARR1 = RESHAPE((/(I, I=1, 9)/), (/3, 3 /))
ARR2 = RESHAPE((/(I, I=9, 1, -1 /), (/3, 3 /))
FLAG = -99

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |  1   4   7 |  ARR2 = |  9   6   3 |  FLAG = | -99 -99 -99 |
!        |  2   5   8 |         |  8   5   2 |         | -99 -99 -99 |
!        |  3   6   9 |         |  7   4   1 |         | -99 -99 -99 |

WHERE (ARR1 > ARR2)
  FLAG = 1
ELSEWHERE (ARR1 == ARR2)
  FLAG = 0
ELSEWHERE
  FLAG = -1
END WHERE

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |  1   4   7 |  ARR2 = |  9   6   3 |  FLAG = | -1  -1   1 |
!        |  2   5   8 |         |  8   5   2 |         | -1   0   1 |
!        |  3   6   9 |         |  7   4   1 |         | -1   1   1 |
```

Related information

- "WHERE construct" on page 121
- "WHERE" on page 493
- "END (Construct)" on page 351, for details on the **END WHERE** statement

END

Purpose

An **END** statement indicates the end of a program unit or procedure.

Syntax

```
>>--END--+----------------------------------------------+--><
         |                                              |
         +--BLOCK DATA--+-------------------+-----------+
         |              +--BLOCK_DATA_name--+           |
         +--FUNCTION--+-----------------+---------------+
         |            +--FUNCTION_name--+               |
         +--MODULE--+---------------+-------------------+
         |          +--MODULE_name--+                   |
         |   (1)                                        |
         +-------PROCEDURE--+------------------+--------+
         |                  +--PROCEDURE_name--+        |
         +--PROGRAM--+----------------+-----------------+
         |           +--PROGRAM_name--+                 |
         +--SUBROUTINE--+-------------------+-----------+
         |              +--SUBROUTINE_name--+           |
         |   (2)                                        |
         +-------SUBMODULE--+------------------+--------+
                            +--SUBMODULE_name--+
```

Notes:

1   Fortran 2008

2   Fortran 2008
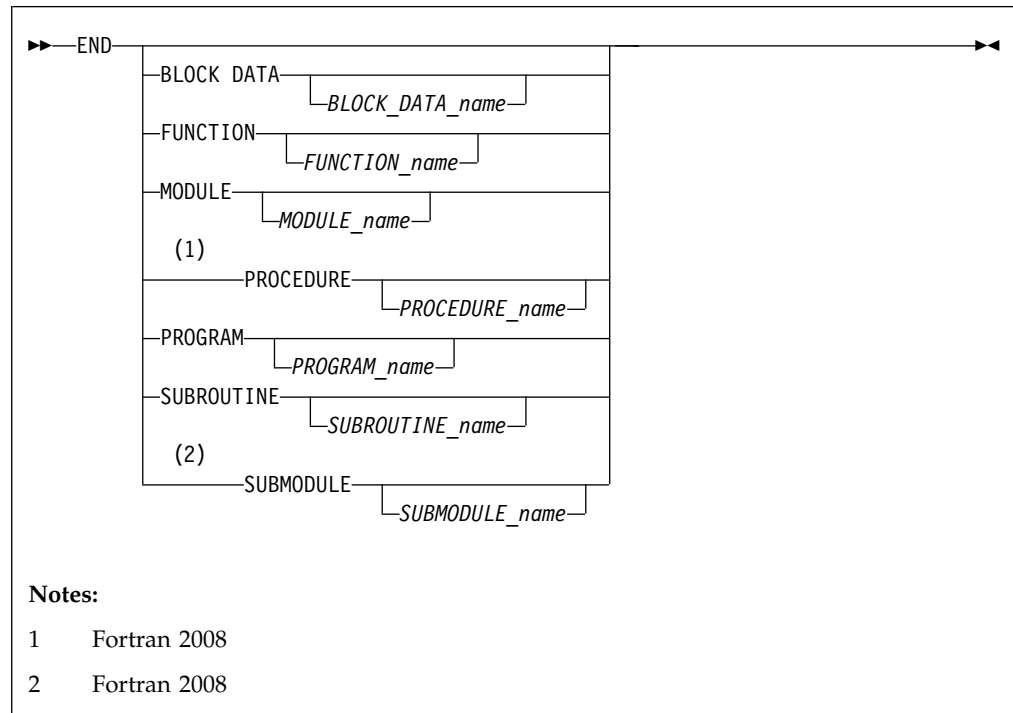
Rules

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

If a name is specified in an **END BLOCK DATA**, **END FUNCTION**, **END MODULE**, **END PROGRAM**, **END SUBROUTINE**, F2008 **END SUBMODULE**, or **END PROCEDURE** F2008 statement, it must be identical to the name that is specified in the corresponding **BLOCK DATA**, **FUNCTION**, **MODULE**, **PROGRAM**, **SUBROUTINE**, F2008 **SUBMODULE**, or **MODULE PROCEDURE** F2008 statement.

For an internal subprogram or module subprogram, you must specify the **FUNCTION** or **SUBROUTINE** keyword on the **END** statement. F2008 In Fortran 2008, you can omit the **FUNCTION** and **SUBROUTINE** keywords on the **END** statements for internal and module subprograms. However, you cannot add a function or subroutine name on the **END** statement when the **FUNCTION** or **SUBROUTINE** keyword is omitted. F2008 For block data program units, external subprograms, the main program, modules, interface bodies, F2008 submodules, and separate module subprograms F2008 , the corresponding keyword is optional.

The **END**, **END FUNCTION**, F2008 **END PROCEDURE** F2008 , **END PROGRAM**, and **END SUBROUTINE** statements are executable statements that can be branched to. In both fixed source form and Fortran 90 free source form

formats, no other statement can follow the **END** statement on the same line. In fixed source form format, you cannot continue a program unit **END** statement, nor can a statement whose initial line appears to be a program unit **END** statement be continued.

The **END** statement of a main program terminates execution of the program. The **END** statement of a function or subroutine has the same effect as a **RETURN** statement. An inline comment can appear on the same line as an **END** statement. Any comment line appearing after an **END** statement belongs to the next program unit.

Examples

```
PROGRAM TEST
  CALL SUB()
  CONTAINS
    SUBROUTINE SUB

    :

    END SUBROUTINE    ! Reference to subroutine name SUB is optional
END PROGRAM TEST
```

Related information

- Chapter 8, "Program units and procedures," on page 151

END (Construct)

Purpose

The **END (Construct)** statement terminates the execution of a construct. The *Construct Termination Statements* table lists the appropriate statement to end each construct.

Table 41. Construct termination statements

Construct	Termination Statement
ASSOCIATE [1]	**END ASSOCIATE**
BLOCK [2]	**END BLOCK**
DO	**END DO**
DO CONCURRENT [2]	
DO WHILE	
FORALL	**END FORALL**
IF	**END IF**
SELECT CASE	**END SELECT**
SELECT TYPE [1]	
WHERE	**END WHERE**
Notes: [1] Fortran 2003 [2] Fortran 2008	

Syntax

```
►►──END──┬─ASSOCIATE────────────────────────────────────────────┬──►◄
         │              ┌────────────────────────────(1)          │
         │              └─ASSOCIATE_construct_name─┘               │
         ├─BLOCK─────────────────────────────────────────────────┤
         │              ┌──────────────────────(2)                │
         │              └─BLOCK_construct_name─┘                   │
         ├─DO──┬──────────────────────┬─────────────────────────┤
         │     └─DO_construct_name──┘                            │
         ├─FORALL─┬──────────────────────────┬──────────────────┤
         │        └─FORALL_construct_name──┘                     │
         ├─IF─┬────────────────────┬─────────────────────────────┤
         │    └─IF_construct_name─┘                              │
         ├─SELECT─┬─────────────────────────────────(3)──────────┤
         │        ├─CASE_construct_name─────────┐                │
         │        └─SELECT_TYPE_construct_name─┘                 │
         └─WHERE─┬───────────────────────┬──────────────────────┘
                 └─where_construct_name─┘
```

Notes:

1   Fortran 2003

2   Fortran 2008
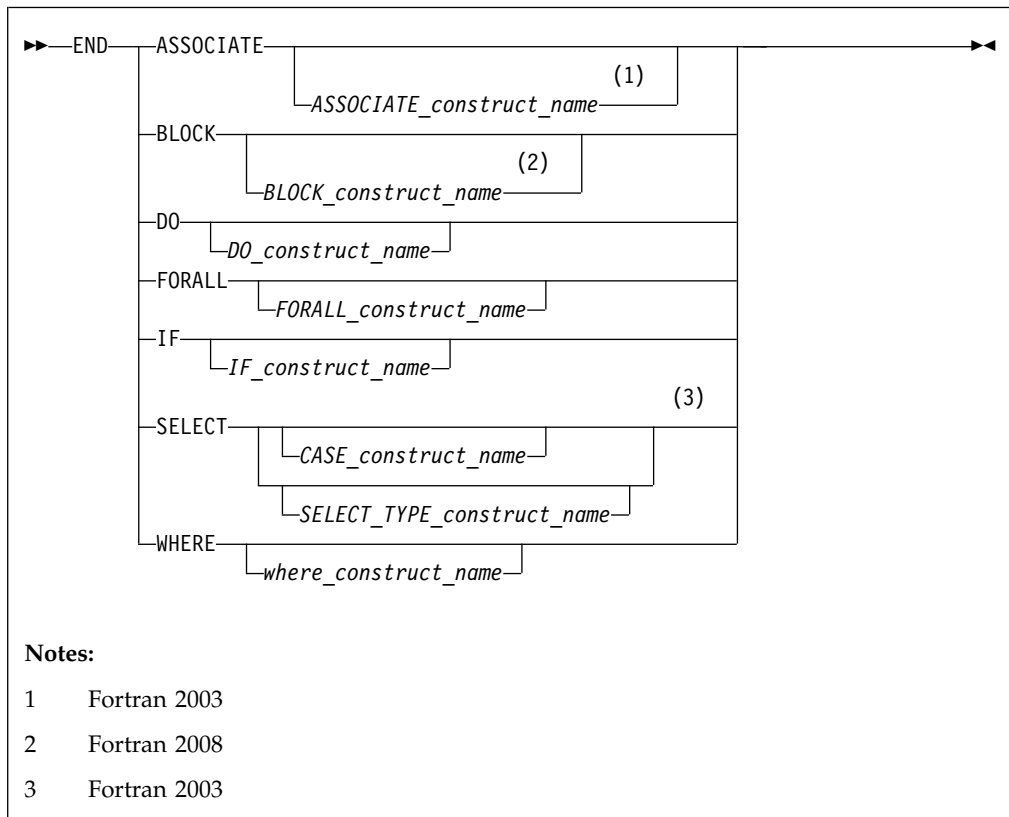
3   Fortran 2003

ASSOCIATE_construct_name **(Fortran 2003)**
> A name that identifies an **ASSOCIATE** construct.

BLOCK_construct_name **(Fortran 2008)**
> A name that identifies a **BLOCK** construct.

DO_construct_name
> A name that identifies a **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct.

FORALL_construct_name
> A name that identifies a **FORALL** construct.

IF_construct_name
> A name that identifies an **IF** construct.

CASE_construct_name
> A name that identifies a **SELECT CASE** construct.

SELECT_TYPE_construct_name **(Fortran 2003)**
> A name that identifies a **SELECT TYPE** construct.

where_construct_name
> A name that identifies a **WHERE** construct.

Rules

If you label the **END DO** statement, you can use it as the terminal statement of a labeled or unlabeled **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct. An **END DO** statement terminates the innermost **DO**, `F2008` **DO**

CONCURRENT `F2008` , or **DO WHILE** construct only. If a **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE** statement does not specify a statement label, the terminal statement of the **DO** or **DO WHILE** construct must be an **END DO** statement.

You can branch from inside or outside of the following constructs to their corresponding **END** statements.

*Table 42. Branch from inside or outside of a construct to its **END** statement*

Construct name	Branch from inside	Branch from outside	Branch target
ASSOCIATE 1	√		**END ASSOCIATE 1**
BLOCK 2	√	√	**END BLOCK 2**
DO	√		**END DO**
DO CONCURRENT 2	√		
DO WHILE	√		
IF 3	√	√	**END IF**
CASE	√		**END SELECT**

Notes:
  1 Fortran 2003
  2 Fortran 2008
  3 In Fortran 95, you cannot branch from outside of an **IF** construct to its **END IF** statement.			

If you specify a construct name on the statement that begins the construct, the **END** statement that terminates the construct must have the same construct name. Conversely, if you do not specify a construct name on the statement that begins the construct, you must not specify a construct name on the **END** statement.

An **END WHERE** statement must not be a branch target statement.

Examples

```
INTEGER X(100,100)
DECR: DO WHILE (I.GT.0)
   ...
  IF (J.LT.K) THEN
   ...
  END IF                ! Cannot reference a construct name
  I=I-1
END DO DECR             ! Reference to construct name DECR mandatory

END
```

The following example shows an invalid use of the *where_construct_name*:

```
BW: WHERE (A /= 0)
  B = B + 1
END WHERE EW    ! The where_construct_name on the END WHERE statement
                ! does not match the where_construct_name on the WHERE
                ! statement.
```

Related information

- Chapter 7, "Execution control," on page 135
- "ASSOCIATE Construct (Fortran 2003)" on page 135
- "BLOCK construct (Fortran 2008)" on page 136

END INTERFACE

Purpose

The **END INTERFACE** statement terminates a procedure interface block.

Syntax
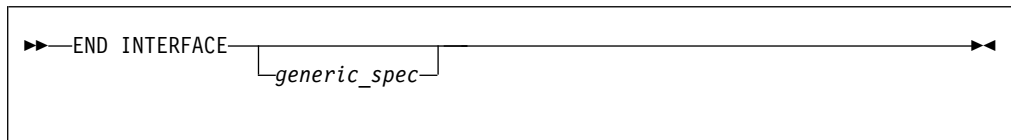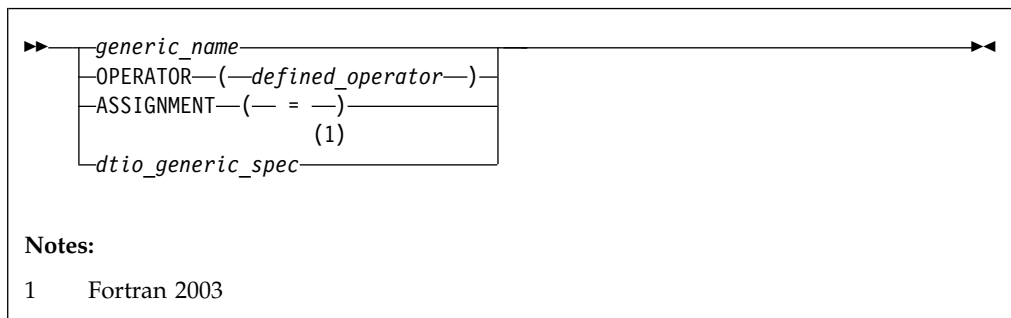
```
►►─── END INTERFACE ──────────────────────────────────────────────►◄
                      └─generic_spec─┘
```

generic_spec

```
►►─────── generic_name ──────────────────────────────────────────►◄
     ├─ OPERATOR ── ( ── defined_operator ── ) ─┤
     ├─ ASSIGNMENT ── ( ── = ── ) ─────────────┤
     │                        (1)               │
     └─ dtio_generic_spec ─────────────────────┘
```

Notes:

1   Fortran 2003

defined_operator
  is a defined unary operator, defined binary operator, or extended intrinsic operator

────────────────────────────── **Fortran 2003** ──────────────────────────────

dtio_generic_spec

```
►►───── READ ── ( ── FORMATTED ── ) ──────────────────────────────►◄
     ├─ READ ── ( ── UNFORMATTED ── ) ─┤
     ├─ WRITE ── ( ── FORMATTED ── ) ──┤
     └─ WRITE ── ( ── UNFORMATTED ── ) ┘
```
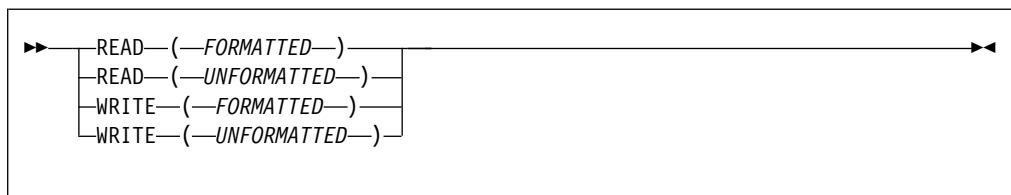
Rules

Each **INTERFACE** statement must have a corresponding **END INTERFACE** statement.

An **END INTERFACE** statement without a *generic_spec* can match any **INTERFACE** statement, with or without a *generic_spec*.

If the *generic_spec* in an **END INTERFACE** statement is a *generic_name*, the *generic_spec* of the corresponding **INTERFACE** statement must be the same *generic_name*.

If the *generic_spec* in an **END INTERFACE** statement is an **OPERATOR**(*defined_operator*), the *generic_spec* of the corresponding **INTERFACE** statement must be the same **OPERATOR**(*defined_operator*).

If the *generic_spec* in an **END INTERFACE** statement is an **ASSIGNMENT**(=), the *generic_spec* for the corresponding **INTERFACE** statement must be the same **ASSIGNMENT**(=).

F2003 If the *generic_spec* in an **END INTERFACE** statement is a *dtio_generic_spec*, the *generic_spec* for the corresponding **INTERFACE** statement must be the same *dtio_generic_spec*. **F2003**

Examples

```
INTERFACE OPERATOR (.DETERMINANT.)
  FUNCTION DETERMINANT (X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE

INTERFACE OPERATOR(.INVERSE.)
  FUNCTION INVERSE(Y)
    INTENT(IN) Y
    REAL Y(50,50), INVERSE
  END FUNCTION
END INTERFACE OPERATOR(.INVERSE.)
```

Related information
- "INTERFACE" on page 405
- "Interface concepts" on page 162

END TYPE

Purpose

The **END TYPE** statement indicates the completion of a derived-type definition.

Syntax

```
►►──END TYPE──────────────────────────────────────────────────────────►◄
              └─type_name─┘
```
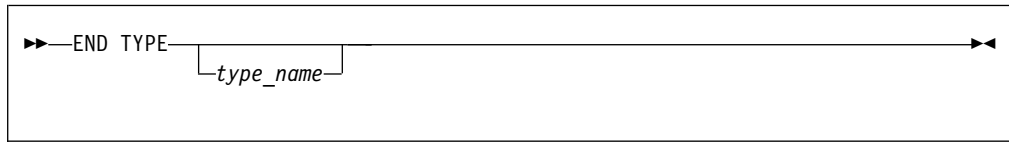
Rules

If *type_name* is specified, it must match the *type_name* in the corresponding **Derived Type**.

If a label is specified on the **END TYPE** statement, the label belongs to the scoping unit of the derived-type definition.

Examples

```
TYPE A
  INTEGER :: B
  REAL :: C
END TYPE A
```

Related information

- Chapter 4, "Derived types," on page 49

ENDFILE

Purpose

The **ENDFILE** statement writes an endfile record as the next record of an external file connected for sequential access. This record becomes the last record in the file.

An **ENDFILE** statement for a file connected for stream access causes the terminal point to become the current file position. File storage units before the current position are considered written, and can be read. You can write additional data to the file by using subsequent stream output statements.

F2003 Execution of an **ENDFILE** statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit. F2003

Syntax

```
►►──ENDFILE──┬──u──────────────────┬──────────────────────────────────►◄
             └─(──position_list──)─┘
```

u is an external unit identifier. The value of *u* must not be an asterisk or a Hollerith constant.

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:

[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. It is one of the following:

- An integer expression whose value is in the range 1 through 2147483647
- `F2008` A NEWUNIT value `F2008`

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

IOMSG= *iomsg_variable* **(Fortran 2003)**

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

is an input/output status specifier that specifies the status of the input/output operation. *ios* is a scalar variable of type **INTEGER(4)** or default integer. When the **ENDFILE** statement finishes executing, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

ERR= *stmt_label*

is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

Rules

IBM extension

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 two **ENDFILE** statements are executed for the same file without an intervening **REWIND** or **BACKSPACE** statement, the second **ENDFILE** statement is ignored.

End of IBM extension

After execution of an **ENDFILE** statement for a file connected for sequential access, a **BACKSPACE** or **REWIND** statement must be used to reposition the file prior to execution of any data transfer input/output statement.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If **IOSTAT=** and **ERR=** are not specified,

- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

Examples

```
ENDFILE 12
ENDFILE (IOSTAT=IOSS,UNIT=11)
```

Related information

- "Conditions and IOSTAT values" on page 226
- Chapter 9, "XL Fortran Input/Output," on page 215
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*

ENTRY

Purpose

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 alternative entry point for an external subprogram or a module subprogram.

Syntax

```
►►──ENTRY──entry_name──┬───────────────────────────┬──────────────────►
                       └─(──┬──────────────────┬──)─┘
                            └─dummy_argument_list─┘

   ┌──────────────────────────────────────────────────┐
►──▼─┬────────────────────────────────────────────┬───┴───────────────►◄
     ├─RESULT──(──result_name──)──────────────────┤
     │        (1)                                  │
     └─BIND──────(──C──┬──────────────────────────┬──)─┘
                       └─,──NAME── = ──binding_label─┘
```

Notes:

1   Fortran 2003
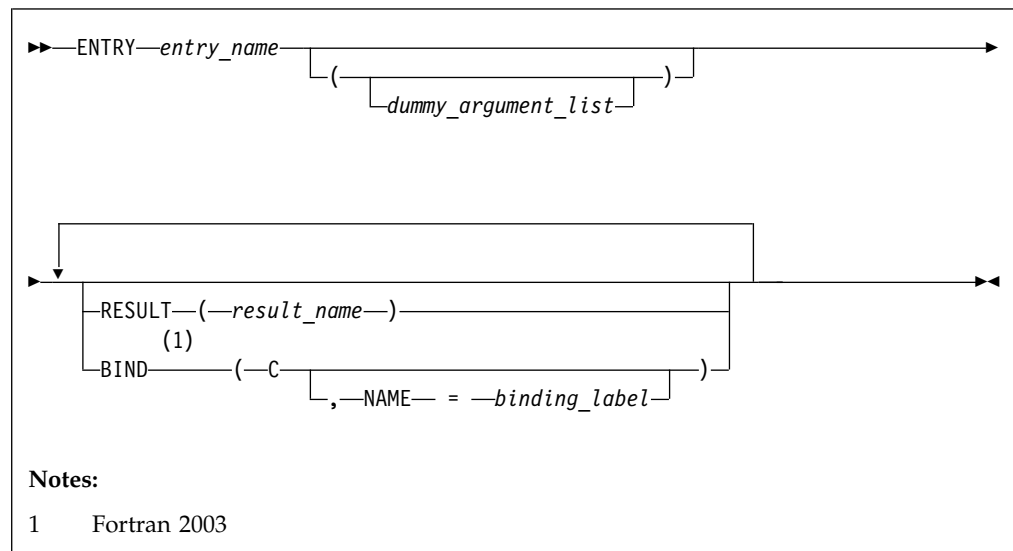
entry_name
> is the name of an entry point in a function subprogram or subroutine subprogram

F2003 *binding_label*
   is a scalar expression for initializing a character F2003

Rules

The **ENTRY** statement cannot appear in a main program, block data program unit, internal subprogram, **IF** construct, **DO** construct, **CASE** construct, derived-type definition, or interface block.

IBM The **ENTRY** statement cannot appear in a **CRITICAL**, **MASTER**, **PARALLEL**, **PARALLEL SECTIONS**, **SECTIONS**, or **SINGLE** construct. IBM

An **ENTRY** statement can appear anywhere after the **FUNCTION** or **SUBROUTINE** statement (and after any **USE** statements) of an external or module subprogram, except in a statement block within a control construct, in a derived-type definition, or in an interface block. **ENTRY** statements are nonexecutable and do not affect control sequencing during the execution of a subprogram.

The result variable is *result_name*, if specified; otherwise, it is *entry_name*. If the characteristics of the **ENTRY** statement's result variable are the same as those of the **FUNCTION** statement's result variable, the result variables identify the same variable, even though they can have different names. Otherwise, they are storage-associated and must be all nonpointer, nonallocatable scalars of intrinsic (noncharacter) type. *result_name* can be the same as the result variable name specified for the **FUNCTION** statement or another **ENTRY** statement.

The result variable cannot be specified in a **COMMON**, **DATA**, integer **POINTER**, or **EQUIVALENCE** statement, nor can it have the **PARAMETER**, **INTENT**, **OPTIONAL**, **SAVE**, or **VOLATILE** attributes. The **STATIC** and **AUTOMATIC** attributes can be specified only when the result variable is not an allocatable object, an array or a pointer, and is not of character or derived type.

If the **RESULT** keyword is specified, the **ENTRY** statement must be within a function subprogram, *entry_name* must not appear in any specification statement in the scope of the function subprogram, and *result_name* cannot be the same as *entry_name*.

A result variable must not be initialized in a type declaration statement or **DATA** statement.

The entry name in an external subprogram is a global entity; an entry name in a module subprogram is not a global entity. An interface for an entry can appear in an interface block only when the entry name is used as the procedure name in an interface body.

At most one **RESULT** clause and at most one **BIND** clause can appear. They can appear in any order.

F2003 The **BIND** keyword implicitly or explicitly defines a binding label which specifies the name by which an entity is accessed from the C programming language. The result variable, if there is a result, must be a scalar that is interoperable. A binding label cannot be specified for a dummy argument. A dummy argument cannot be zero-sized. A dummy argument for a procedure with the **BIND** attribute must have interoperable types and type parameters, and cannot have the **ALLOCATABLE**, **OPTIONAL**, or **POINTER** attribute. F2003

In a function subprogram, *entry_name* identifies a function and can be referenced as a function from the calling procedure. In a subroutine subprogram, *entry_name* identifies a subroutine and can be referenced as a subroutine from the calling procedure. When the reference is made, execution begins with the first executable statement following the **ENTRY** statement.

The result variable must be defined before exiting from the function, if the function is invoked through that **ENTRY** statement.

A name in the *dummy_argument_list* must not appear in the following places:
- 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 order, number, type, and kind type parameters of the dummy arguments can differ from those of the **FUNCTION** or **SUBROUTINE** statement, or other **ENTRY** statements.

Suppose a dummy argument is used in a specification expression to specify an array bound or character length of an object. You can only specify the object in a statement that is executed during a procedure reference if the dummy argument is present and appears in the dummy argument list of the procedure name referenced.

F2008 **Note:** The **ENTRY** statement is marked as obsolescent in Fortran 2008 and later language standards. A warning message is generated if you use an **ENTRY** statement when -qlanglvl=2008pure is specified. Instead, you can use a module containing the private data item, with a module procedure for each entry point and the shared code in a private module procedure. **F2008**

Recursion

An **ENTRY** statement can reference itself directly only if the subprogram statement specifies **RECURSIVE** and the **ENTRY** statement specifies **RESULT**. The entry procedure then has an explicit interface within the subprogram. The **RESULT** clause is not required for an entry to reference itself indirectly.

Elemental subprograms can have **ENTRY** statements, but the **ENTRY** statement cannot have the **ELEMENTAL** prefix. The procedure defined by the **ENTRY** statement is elemental if the **ELEMENTAL** prefix is specified in the **SUBROUTINE** or **FUNCTION** statement.

In a recursive function, if *entry_name* is of type character, its length cannot be represented by an asterisk (*, meaning assumed or specified elsewhere).

IBM You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if a procedure specifies either the **RECURSIVE** or **RESULT** keyword. **IBM**

Examples

```
RECURSIVE FUNCTION FNC() RESULT (RES)

  ⋮
  ENTRY ENT () RESULT (RES)              ! The result variable name can be
                                         ! the same as for the function

  ⋮
END FUNCTION
```

Related information

* "FUNCTION" on page 379
* "SUBROUTINE" on page 468
* "Recursion" on page 208
* "Dummy arguments" on page 194
* **-qrecur** option in the *XL Fortran Compiler Reference*

ENUM/END ENUM (Fortran 2003)

Purpose

You can specify an **ENUM** statement to define and group a set of named integer constants. The named integer constants in an **ENUM** statement are called enumerators.

Syntax

To define an enumerator, you must use an enumeration construct:

Enumeration construct

▶▶──ENUM, BIND(C)──▶◀

▶▶──*enumeration_block*───▶◀

▶▶──ENDENUM──▶◀

---

enumeration_block

```
                        ┌──────,──────┐
▶▶──ENUMERATOR──┬────┬──▼─named_constant─┬──────────────────┬──┴──▶◀
                └─::─┘                   └─=─scalar_int_exp─┘
```
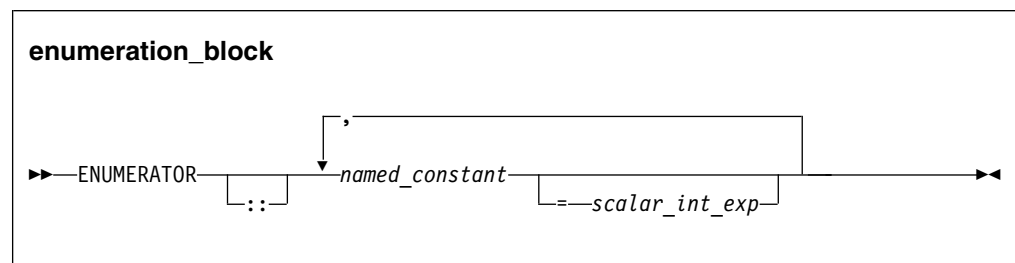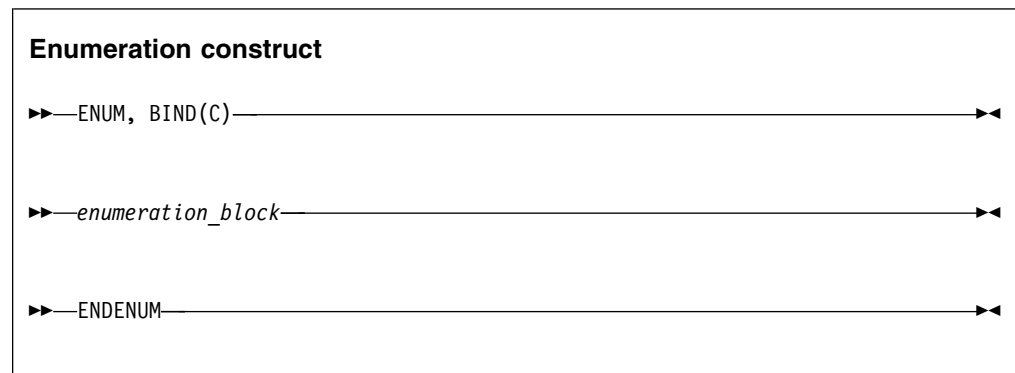
If you want to specify an enumerator with a *scalar_int_exp*, you must also specify a double colon seperator (::).

Rules

If you specify a scalar integer constant expression, the value of the enumerator is the result of the scalar integer constant expression.

IBM You can use a scalar logical constant expression only if you compile with `-qintlog`. **IBM**

If you do not specify a scalar integer constant expression and the enumerator is first in the *enumeration_block*, the value of the enumerator is 0.

If you do not specify a scalar integer constant expression and the enumerator is after another enumerator in the *enumeration_block*, the value is one greater than the value of the preceding enumerator.

You can set the kind type parameter of an enumerator using the **-qenum** option. If you do not specify **-qenum**, the default kind for an enumerator is 4.

Examples

The following example uses the **ENUM** statement in different ways to define enumerators.

```
enum, bind(c)

    enumerator :: red =1, blue, black =5
    enumerator yellow
    enumerator gold, silver, bronze
    enumerator :: purple
    enumerator :: pink, lavender

endenum
```

The values of these enumerators are: red = 1, blue = 2, black = 5, yellow = 6 , gold = 7, silver = 8, bronze = 9, purple = 10, pink = 11, lavender = 12.

If you supply an initial value for an enumerator, then a *::* is required in the **ENUMERATOR** statement. The *red* and *black* enumerators in the list are initialized with a scalar integer constant expression.

The *::* is optional in an enumerator definition when scalar integer constant expressions are not used to initialize any of the enumerators in the list of enumerators being declared:

- In the second and third enumerator definitions, the *::* is not necessary as *yellow*, *gold*, *silver*, and *bronze* are not initialized with a scalar integer constant expression.
- The fourth and fifth enumerator definitions show that *::* can be used even when *purple* is not initialized with a scalar integer constant expression.

Related information
- "PARAMETER" on page 424

EQUIVALENCE

Purpose

The **EQUIVALENCE** statement specifies that two or more objects in a scoping unit are to share the same storage.

Syntax

```
                             ,
                           ┌───┐
>>──EQUIVALENCE──▼──(──equiv_object──,──equiv_object_list──)────────><
```
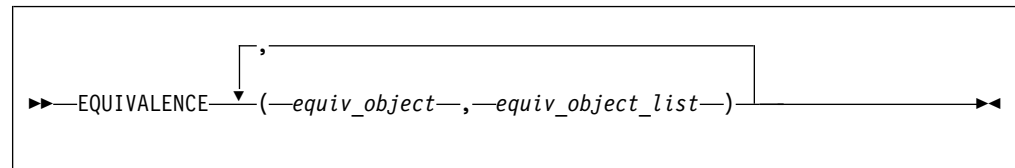
equiv_object
> is a variable name, array element, or substring. Any subscript or substring expression must be an integer constant expression. A substring cannot have a length of zero.

Rules

equiv_object must not be a target, pointer, dummy argument, function name, pointee, entry name, result name, structure component, named constant, automatic data object, allocatable object, object of nonsequence derived type, object of sequence derived type that contains a pointer or allocatable component, or a subobject of any of these.

F2003 Variables with the **BIND** attribute, or variables that are members of a common block with the **BIND** attribute must not be objects in an **EQUIVALENCE** statement. F2003

Because all items named within a pair of parentheses have the same first storage unit, they become associated. This is called *equivalence association*. It may cause the association of other items as well.

You can specify default initialization for a storage unit that is storage associated. However, the objects or subobjects supplying the default initialization must be of the same type. They must also be of the same type parameters and supply the same value for the storage unit.

If you specify an array element in an **EQUIVALENCE** statement, the number of subscript quantities cannot exceed the number of dimensions in the array. If you specify a multidimensional array using an array element with a single subscript *n*, the *n* element in the array's storage sequence is specified. In all other cases, XL Fortran replaces any missing subscript with the lower bound of the corresponding dimension of the array. A nonzero-sized array without a subscript refers to the first element of the array.

If *equiv_object* is of derived type, it must be of a sequence derived type.

You can equivalence an object of sequence derived type with any other object of sequence derived type or intrinsic data type provided that the object is allowed in an **EQUIVALENCE** statement.

In XL Fortran, associated items can be of any intrinsic type or of sequence derived type. If they are, the **EQUIVALENCE** statement does not cause type conversion.

The lengths of associated items do not have to be equal.

Any zero-sized items are storage-associated with one another and with the first storage unit of any nonzero-sized sequences.

An **EQUIVALENCE** statement cannot associate the storage sequences of two different common blocks. 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 can cause names not in common blocks to share storage with a name in a common block using the **EQUIVALENCE** statement.

F2003 If you specify that an object declared by an **EQUIVALENCE** group has the **PROTECTED** attribute, all objects specified in that **EQUIVALENCE** group must have the **PROTECTED** attribute. F2003

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, 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.

Examples

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

Association of storage units:

```
          |      |      |      |      |      |      |      |
Array A:   |      |      |      A(1)  |     A(2)  |     A(3)  |
Array B:  | B(1) | B(2) | B(3) | B(4) | B(5) |
```

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

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

Association of storage units:

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

Because XL Fortran associates both A and B with C, A and B become associated with each other, and they all have the automatic storage class.

```
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)
```

Related information
- "Storage classes for variables (IBM extension)" on page 28
- "Definition status of variables" on page 21

ERROR STOP (Fortran 2008)

Purpose

The **ERROR STOP** statement initiates error termination of a program, which terminates the execution of the program. If a *stop_code* is specified, the keyword "ERROR STOP" followed by the *stop_code* is printed to ERROR_UNIT.
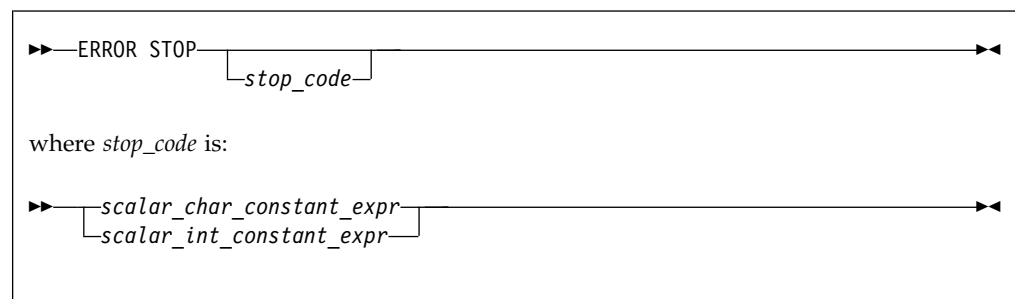
Syntax

```
>>─ERROR STOP──────────────────────────────────────────><
              └─stop_code─┘
```

where *stop_code* is:

```
>>──┬─scalar_char_constant_expr─┬──────────────────────><
    └─scalar_int_constant_expr──┘
```

scalar_char_constant_expr
> is a scalar character constant expression

scalar_int_constant_expr
> is a scalar integer constant expression

Rules

When an ERROR STOP statement is executed, a system return code is supplied and an error message is printed to ERROR_UNIT, depending on whether the *stop_code* is specified:

- If the *stop_code* is *scalar-char-constant-expr*, the system return code is 1. The keyword "ERROR STOP" followed by the *stop_code* is printed.
- If the *stop_code* is *scalar-int-constant-expr*, XL Fortran sets the system return code to **MOD** (*stop_code*, 256). The keyword "ERROR STOP" followed by the *stop_code* is printed.
- If nothing is specified, the system return code is 1. No error message is printed.

You cannot use an ERROR STOP statement as the labeled statement that terminates a **DO** construct.

Examples

The following example shows how ERROR STOP statements are used:

```fortran
PROGRAM p
  INTEGER, SAVE :: s = -1
  INTEGER, SAVE :: arr(3) = -1

  ! If the initialization for s is wrong, the error
  ! message "ERROR STOP Initial value wrong!" is printed.
  ! The system return code is 1.
  IF (s .NE. -1) ERROR STOP "Initial value wrong!"

  ! If the initialization for arr is wrong, no message is printed.
  ! The system return code is 1.
  IF (ANY(arr .NE. -1)) ERROR STOP

  s = 1
  arr = 1

  ! If the value for s is not 1, the error message "ERROR STOP 127" is printed.
  ! The system return code is 127.
  IF (s .NE. 1) ERROR STOP 127

  ! If the value for arr is not 1, the error message "ERROR STOP 0" is printed.
  ! The system return code is 0.
  IF (ANY(arr .NE. 1)) ERROR STOP 0

  STOP "Good!"
END PROGRAM p
```

Related information
- "STOP" on page 464
- "Non-finalized entities" on page 69

EXIT

Purpose

The **EXIT** statement terminates execution of a **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** construct before the construct terminates all of its iterations. In addition, it can be used to terminate execution of a specified construct that is not **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE**.

Syntax

```
►►──EXIT──┬────────────────┬──────────────────────────────────►◄
          └─construct_name─┘
```
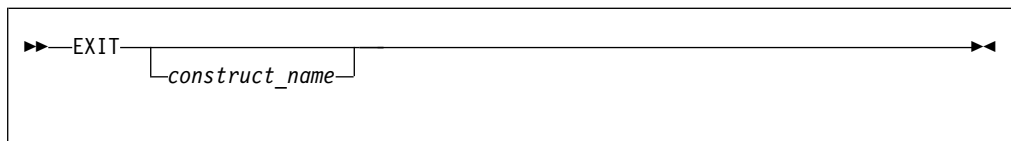
construct_name
> The name of a construct.
>
> It can be one of the following constructs:
> - ASSOCIATE
> - `F2008` BLOCK `F2008`
> - DO
> - `F2008` DO CONCURRENT `F2008`

- DO WHILE
- IF
- SELECT CASE
- SELECT TYPE

Rules

If *construct_name* is specified, the **EXIT** statement must be within the construct specified by *construct_name*. If *construct_name* is not specified, the **EXIT** statement must be within the range of at least one **DO** or **DO WHILE** construct.

If *construct_name* is specified, the **EXIT** statement belongs to the construct specified by *construct_name*. If *construct_name* is not specified, the **EXIT** statement belongs to the **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE** construct that immediately surrounds it.

`F2008` An **EXIT** statement must not be placed within a **CRITICAL** or **DO CONCURRENT** construct if it belongs to that construct or an outer construct. `F2008`

If an **EXIT** statement belongs to a **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE** construct, execution of the **EXIT** statement causes the construct to become inactive. If the **EXIT** statement is nested in any other **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE** constructs, they also become inactive. Any **DO** variable present retains its last defined value. If the **DO** construct has no construct control, it will iterate infinitely unless it becomes inactive. The **EXIT** statement can be used to make the construct inactive.

`F2008` If an **EXIT** statement belongs to a construct that is not **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE**, execution of the **EXIT** statement terminates execution of the construct. Any **DO**, `F2008` **DO CONCURRENT** `F2008` , or **DO WHILE** loops contained within the construct become inactive. `F2008`

An **EXIT** statement can have a statement label; it cannot be used as the labeled statement that terminates a construct.

Examples

Example 1: The following example illustrates the usage of the **EXIT** statement in the **DO** and **DO WHILE** statements:

```
      LOOP1: DO I = 1, 20
         N = N + 1
10       IF (N > NMAX) EXIT LOOP1        ! EXIT from LOOP1

         LOOP2: DO WHILE (K==1)
            KMAX = KMAX - 1
20          IF (K > KMAX) EXIT           ! EXIT from LOOP2
         END DO LOOP2

         LOOP3:  DO J = 1, 10
            N = N + 1
30          IF (N > NMAX) EXIT LOOP1     ! EXIT from LOOP1
            EXIT LOOP3                   ! EXIT from LOOP3
         END DO LOOP3

      END DO LOOP1
```

`F2008`

Example 2: The following example shows how the **EXIT** statement is used to terminate execution of a **BLOCK** construct:

```
a : BLOCK
  DO i = 1, num_in_set
    IF (X == a(i)) EXIT a       ! EXIT from the a BLOCK construct
  END DO
  CALL r
END BLOCK a
```

F2008

Related information

- "ASSOCIATE (Fortran 2003)" on page 292
- "BLOCK (Fortran 2008)" on page 299
- "CASE" on page 305
- "SELECT CASE" on page 459
- "DO construct" on page 138
- "DO WHILE construct" on page 142
- "DO CONCURRENT construct (Fortran 2008)" on page 142
- "IF construct" on page 143
- "SELECT TYPE construct (Fortran 2003)" on page 147

EXTERNAL

Purpose

The **EXTERNAL** attribute specifies that a name represents an external procedure, a dummy procedure, or a block data program unit. A procedure name with the **EXTERNAL** attribute can be used as an actual argument.

Syntax

```
>>--EXTERNAL--+------+---name_list----------------------------><
              |      |
              +- :: -+
```
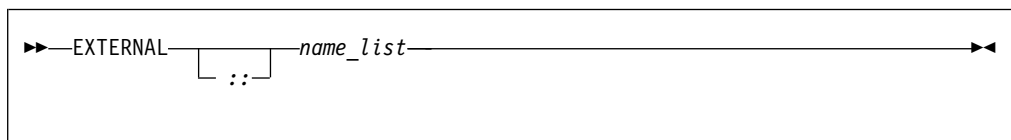
name   is the name of an external procedure, dummy procedure, or **BLOCK DATA** program unit

Rules

If an external procedure name or dummy argument name is used as an actual argument, it must be declared with the **EXTERNAL** attribute or by an interface block in the scoping unit, but may not appear in both.

If an intrinsic procedure name is specified with the **EXTERNAL** attribute in a scoping unit, the name becomes the name of a user-defined external procedure. Therefore, you cannot invoke that intrinsic procedure by that name from that scoping unit.

You can specify a name to have the **EXTERNAL** attribute appear only once in a scoping unit.

A name in an **EXTERNAL** statement must not also be specified in a procedure declaration statement or as a specific procedure name in an interface block in the scoping unit.

Table 43. Attributes compatible with the EXTERNAL attribute

CONTIGUOUS **1**	PRIVATE
OPTIONAL	PUBLIC
Note: **1** Fortran 2008	

Examples

```
PROGRAM MAIN
  EXTERNAL AAA
  CALL SUB(AAA)          ! Procedure AAA is passed to SUB
END

SUBROUTINE SUB(ARG)
  CALL ARG()             ! This results in a call to AAA
END SUBROUTINE
```

Related information

- "Procedures as dummy arguments" on page 205
- Item 4 under "Compatibility across standards" on page 1051

FLUSH (Fortran 2003)

Purpose

The **FLUSH** statement makes data written to an external file available to other processes, or causes data placed in an external file by means other than Fortran to be available to a **READ** statement.

Syntax

```
►►──FLUSH──┬─u──────────┬──────────────────────────────────────►◄
           └─flush_list─┘
```
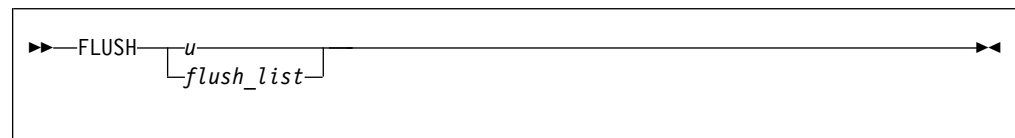
u   is an integer scalar expression which has one of the following values:

- A value in the range 1 through 2147483647
- **F2008** A NEWUNIT value **F2008**

This unit references an external file. The value of the integer scalar expression must not be an asterisk or a Hollerith constant.

flush_list

a list of specifiers that must contain **UNIT=**, and can also contain one of each of the following specifiers:

- **[UNIT=]** specifies the external file as an integer scalar expression which has one of the following values:
  - A value in the range 1 through 2147483647
  - **F2008** A NEWUNIT value **F2008**

The value of the integer scalar expression must not be an asterisk or a Hollerith constant.

- **ERR=***stmt_label* is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Inclusion of the **ERR=** specifier suppresses error messages. *stmt_label* must be the statement label of a branch target statement that appears in the same scoping unit as the **FLUSH** statement.

- **IOMSG=***iomsg_variable* is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:
  - If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
  - If no such condition occurs, the value of the variable is unchanged.

- **IOSTAT=***ios* specifies the status of the flush operation as a scalar variable of type **INTEGER**. When execution of the flush statement completes, **ios** is:
  - A zero value if no error condition occurs.
  - A positive value if an error occurs.
  - A negative value if the device is not seekable such as a tape or TTY and the most recent data transfer operation was input.

  Inclusion of the **IOSTAT** specifier suppresses error messages. If the program encounters a severe error, the value of *ios* is 200.

If you do not specify **ERR** or **IOSTAT**, the program terminates on encountering a severe error.

Rules

The **FLUSH** statement must not appear in a pure subprogram.

A **FLUSH** statement has no effect on file position.

The **buffering** run-time option does not affect the execution of the **FLUSH** statement.

Examples

Example 1:

In the following example a data file written by a Fortran program is read by a C routine. The program specifies a **FLUSH** statement for the buffered I/O.

```
! The following Fortran program writes data to an external file.
subroutine process_data()
  integer data(10)
  external read_data

  data = (/(i,i=1,10)/)
  open(50, file="data_file")
  write(50, *) data                ! write data to an external file
  flush(50)                        ! since Fortran I/O is buffered, a FLUSH
                                   ! statement is needed for the C routine to
                                   ! to read the data
```

```
      call read_data(10)              ! call C routine to read the file
   end subroutine

   /* The following C routine reads data from the external file. */
   void read_data(int *sz) {

   #include < stdio.h>
   #include < stdlib.h>
   int *data, i;
     FILE  *fp;

     data = (int *) malloc((*sz)*sizeof(int));
     fp = fopen("data_file", "r");
     for (i=0; i<*sz-1; i++) {
       fscanf(fp, "%d", &dat5[i]);
     }
   }
```

Related information

- Chapter 9, "XL Fortran Input/Output," on page 215
- "flush_(lunit)" on page 974
- **Flushing I/O buffers** in the *XL Fortran Optimization and Programming Guide*

FORALL

Purpose

The **FORALL** statement performs assignment to groups of subobjects, especially array elements. Unlike the **WHERE** statement, assignment can be performed on an elemental level rather than on an array level. The **FORALL** statement also allows pointer assignment.

Syntax

```
►►──FORALL──forall_header──forall_assignment────────────────────────────►◄
```

forall_header

```
►►──(──┬─────────────────────┬──forall_triplet_spec_list──┬──────────────────────┬──)──►◄
       │          (1)         │                            └─,──scalar_mask_expr──┘
       └─type_spec──::────────┘

   Notes:
   1      Fortran 2008
```

forall_triplet_spec

```
►►──index_name── = ──subscript── : ──subscript──┬───────────────┬──►◄
                                                 └─ : ──stride──┘
```

forall_assignment
> is either *assignment_statement* or *pointer_assignment_statement*

`F2008` *type_spec*
> specifies an integer type `F2008`

scalar_mask_expr
> is a scalar logical expression

subscript, stride
> are each scalar integer expressions

Rules

Only pure procedures can be referenced in the mask expression of *forall_header* and in a *forall_assignment* (including one referenced by a defined operation, assignment, or finalization).

index_name must be a scalar integer variable. It is also a statement entity; that is, it does not affect and is not affected by other entities in the scoping unit.

`F2008`

You must explicitly declare the **FORALL** index variables in the following cases:
- The value range of *index_name* exceeds the range of the default integer type.
- The **IMPLICIT NONE** statement is in effect. See Example 2.

You can specify *type_spec* to declare *index_name* within the scope of the **FORALL** statement. If you specify *type_spec* in *forall_header*, you can reuse any accessible identifier for *index_name*.

`F2008`

In *forall_triplet_spec_list*, neither a *subscript* nor a *stride* can contain a reference to any *index_name* in the *forall_triplet_spec_list*. Evaluation of any expression in *forall_header* must not affect evaluation of any other expression in *forall_header*.

Given the *forall_triplet_spec*
```
index1 = s1:s2:s3
```

the maximum number of index values is determined by:
```
max = INT((s2-s1+s3)/s3)
```

If the stride (*s3* above) is not specified, a value of 1 is assumed. If *max* ≤ 0 for any index, *forall_assignment* is not executed. For example,
```
index1 = 2:10:3     !  The index values are 2,5,8.
                       max = INT((10-2+3)/3) = 3.

index2 = 6:2:-1     !  The index values are 6,5,4,3,2.
index2 = 6:2        !  No index values.
```

If the mask expression is omitted, a value of `.TRUE.` is assumed.

No atomic object can be assigned to more than once. Assignment to a nonatomic object assigns to all subobjects or associates targets with all subobjects.

Examples

Example 1

```
INTEGER A(1000,1000), B(200)
I=17
FORALL (I=1:1000,J=1:1000,I.NE.J) A(I,J)=A(J,I)
PRINT *, I    ! The value 17 is printed because the I
              ! in the FORALL has statement scope.
FORALL (N=1:200:2) B(N)=B(N+1)
END
```

Example 2

```
IMPLICIT NONE
INTEGER, PARAMETER :: limit=60000
INTEGER(1) :: flag(limit)
flag=0
FORALL (INTEGER(4) :: i=2:limit:2) flag(i)=1
PRINT *, flag(limit)
END
```

Related information

- "Intrinsic assignment" on page 117
- "Data pointer assignment" on page 130
- "FORALL construct" on page 127
- "INDEPENDENT" on page 521
- "Statement and construct entity" on page 155

Interpreting the FORALL statement

1. Evaluate the *subscript* and *stride* expressions for each *forall_triplet_spec* in any order. All possible pairings of *index_name* values form the set of combinations. For example, given the following statement:

   ```
   FORALL (I=1:3,J=4:5) A(I,J) = A(J,I)
   ```

   The set of combinations of I and J is:

   {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}

   The **-1** and **-qnozerosize** compiler options do not affect this step.

2. Evaluate the *scalar_mask_expr* for the set of combinations, in any order, producing a set of active combinations (those for which *scalar_mask_expr* evaluated to .TRUE.). For example, if the mask (I+J.NE.6) is applied to the above set, the set of active combinations is:

   {(1,4),(2,5),(3,4),(3,5)}

3. For *assignment_statement*, evaluate, in any order, all values in the right-hand side *expression* and all subscripts, strides, and substring bounds in the left-hand side *variable* for all active combinations of *index_name* values.

   For *pointer_assignment*, determine, in any order, what will be the targets of the pointer assignment and evaluate all subscripts, strides, and substring bounds in the pointer for all active combinations of *index_name* values. Whether or not the target is a pointer, the determination of the target does not include evaluation of its value.

4. For *assignment_statement*, assign, in any order, the computed *expression* values to the corresponding *variable* entities for all active combinations of *index_name* values.

   For *pointer_assignment*, associate, in any order, all targets with the corresponding pointer entities for all active combinations of *index_name* values.

Loop parallelization

The **FORALL** statement and **FORALL** construct are designed to allow for parallelization of assignment statements. When executing an assignment statement in a **FORALL**, the assignment of an object will not interfere with the assignment of another object. In the next example, the assignments to elements of A can be executed in any order without changing the results:

```
FORALL (I=1:3,J=1:3) A(I,J)=A(J,I)
```

--- **IBM extension** ---

The **INDEPENDENT** directive asserts that each iteration of a **DO** loop or each operation in a **FORALL** statement or **FORALL** construct can be executed in any order without affecting the semantics of the program. The operations in a **FORALL** statement or **FORALL** construct are defined as:

- The evaluation of *mask*
- The evaluation of the right-hand side and/or left-hand side indexes
- The evaluation of assignments

Thus, the following loop,

```
      INTEGER, DIMENSION(2000) :: a
!IBM* INDEPENDENT
      FORALL (i=1:1999:2) a(i) = a(i+1)
```

is semantically equivalent to the following array assignment:

```
      INTEGER, DIMENSION(2000) :: A
      A(1:1999:2) = A(2:2000:2)
```

> **Tip:**
>
> If it is possible and beneficial to make a specific **FORALL** parallel, specify the **INDEPENDENT** directive before the **FORALL** statement. Because XL Fortran may not always be able to determine whether it is legal to parallelize a **FORALL**, the **INDEPENDENT** directive provides an assertion that it is legal.

--- **End of IBM extension** ---

FORALL (Construct)

Purpose

The **FORALL (Construct)** statement is the first statement of the **FORALL** construct.
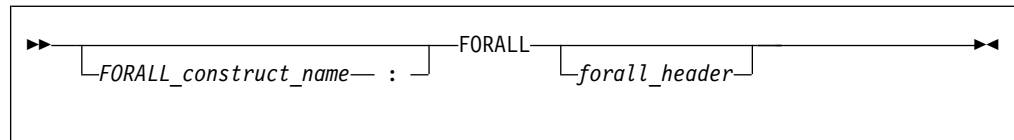
Syntax

```
>>─┬──────────────────────────────┬──FORALL─┬─────────────────┬──><
   └─FORALL_construct_name─ : ─────┘         └─forall_header───┘
```
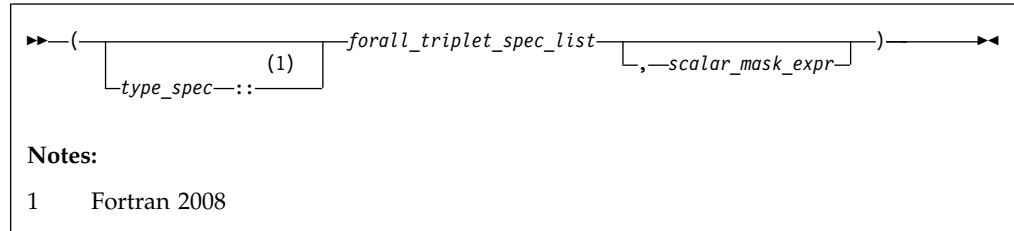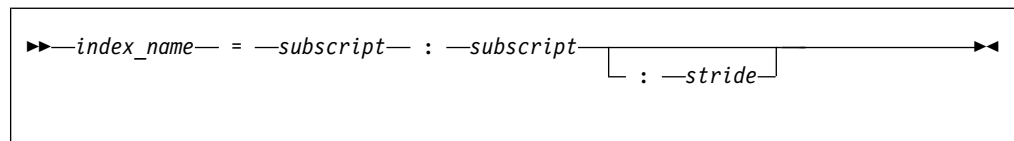
forall_header

```
>>─(─┬──────────────────────┬──forall_triplet_spec_list─┬──────────────────────┬──)──><
     │              (1)      │                           └─,─scalar_mask_expr────┘
     └─type_spec─::──────────┘
```

Notes:

1   Fortran 2008

forall_triplet_spec

```
>>──index_name─ = ─subscript─ : ─subscript─┬───────────────┬──><
                                           └─ : ─stride─────┘
```

F2008 *type_spec*
        specifies an integer type **F2008**

scalar_mask_expr
        is a scalar logical expression

subscript, stride
        are both scalar integer expressions

Rules

Any procedures that are referenced in the mask expression of *forall_header*
(including one referenced by a defined operation or assignment) must be pure.

The *index_name* must be a scalar integer variable. The scope of *index_name* is the
whole **FORALL** construct.

F2008

You must explicitly declare the **FORALL** index variables in the following cases:
- The value range of *index_name* exceeds the range of the default integer type.
- The **IMPLICIT NONE** statement is in effect.

You can specify *type_spec* to declare *index_name* within the scope of the **FORALL**
construct. If you specify *type_spec* in *forall_header*, you can reuse any accessible
identifier for *index_name*.

F2008

In *forall_triplet_spec_list*, neither a *subscript* nor a *stride* can contain a reference to any *index_name* in the *forall_triplet_spec_list*. Evaluation of any expression in *forall_header* must not affect evaluation of any other expression in *forall_header*.

Given the following *forall_triplet_spec*:

```
index1 = s1:s2:s3
```

The maximum number of index values is determined by:

```
max = INT((s2-s1+s3)/s3)
```

If the stride (*s3* above) is not specified, a value of 1 is assumed. If *max* ≤ 0 for any index, *forall_assignment* is not executed. For example:

```
index1 = 2:10:3    !  The index values are 2,5,8.
                   !  max = floor(((10-2)/3)+1) = 3.

index2 = 6:2:-1    !  The index values are 6,5,4,3,2.
index2 = 6:2       !  No index values.
```

If the mask expression is omitted, a value of .TRUE. is assumed.

Examples

```
POSITIVE: FORALL (X=1:100,A(X)>0)
  I(X)=I(X)+J(X)
  J(X)=J(X)-I(X+1)
END FORALL POSITIVE
```

Related information

- "END (Construct)" on page 351
- "FORALL construct" on page 127
- "Statement and construct entity" on page 155

FORMAT

Purpose

The **FORMAT** statement provides format specifications for input/output statements.

Syntax

```
►►──FORMAT──(─────────────────────)──────────────────────────►◄
              └─format_item_list─┘
```

format_item

```
                        data_edit_desc
         r
      control_edit_desc
                             (  format_item_list  )
         r
      char_string_edit_desc
```
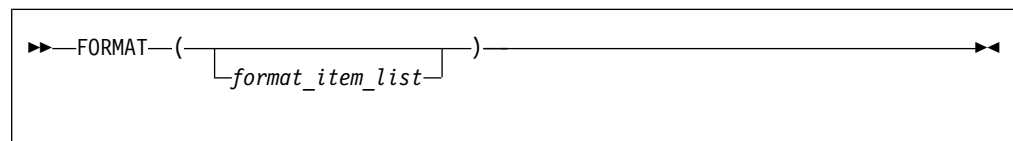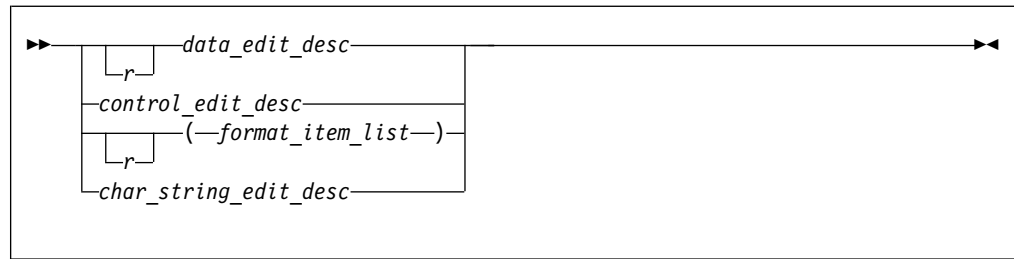
>   *r*   is an unsigned, positive, integer literal constant that cannot specify a kind type parameter, or it is a scalar integer expression enclosed by angle brackets (< and >). It is called a repeat specification. It specifies the number of times to repeat the *format_item_list* or the *data_edit_desc*. The default is 1.

data_edit_desc
>   is a data edit descriptor

control_edit_desc
>   is a control edit descriptor

char_string_edit_desc
>   is a character string edit descriptor

Rules

When a format identifier in a formatted **READ**, **WRITE**, or **PRINT** statement is a statement label or a variable that is assigned a statement label, the statement label identifies a **FORMAT** statement.

The **FORMAT** statement must have a statement label. **FORMAT** statements cannot appear in block data program units, interface blocks, the scope of a module F2008 or submodule F2008 , or derived-type definitions.

Commas separate edit descriptors. You can omit the comma between a **P** edit descriptor and an **F**, **E**, **EN**, **ES**, **D**, **G**, or **Q** (both extended precision and character count) edit descriptor immediately following it, before a slash edit descriptor when the optional repeat specification is not present, after a slash edit descriptor, and before or after a colon edit descriptor.

FORMAT specifications can also be given as character expressions in input/output statements.

XL Fortran treats uppercase and lowercase characters in format specifications the same, except in character string edit descriptors.

Examples

```
CHARACTER*32 CHARVAR
CHARVAR="('integer: ',I2,'  binary: ',B8)"  ! Character format
M = 56                                      ! specification
J = 1                                       !     OUTPUT:
X = 2355.95843                              !
WRITE (6,770) M,X                           !  56    2355.96
WRITE (6,CHARVAR) M,M                       ! integer: 56
                                            ! binary: 00111000
WRITE (6,880) J,M                           !  1
```

```
                                                       ! 56
      770   FORMAT(I3, 2F10.2)
      880   FORMAT(I<J+1>)
            END
```

Related information

- Chapter 10, "Input/Output formatting," on page 237
- "PRINT" on page 430
- "READ" on page 441
- "WRITE" on page 495

Character format specification

When a format identifier 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.

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 without affecting the format specification.

Variable format expressions (IBM extension)

Wherever an integer constant is required by an edit descriptor, you can specify an integer expression in a **FORMAT** statement. The integer expression must be enclosed by angle brackets (< and >). You cannot use a sign outside of a variable format expression. The following are valid format specifications:

```
      WRITE(6,20) INT1
20    FORMAT(I<MAX(20,5)>)

      WRITE(6,FMT=30) INT2, INT3
30    FORMAT(I<J+K>,I<2*M>)
```

The integer expression can be any valid Fortran expression, including function calls and references to dummy arguments, with the following restrictions:

- Expressions cannot be used with the **H** edit descriptor
- Expressions cannot contain graphical relational operators.

The value of the expression is reevaluated each time an input/output item is processed during the execution of the **READ**, **WRITE**, or **PRINT** statement.

Examples

```
      CHARACTER*32 CHARVAR
      CHARVAR="('integer: ',I2,'  binary: ',B8)"  ! Character format
      M = 56                                       ! specification
      J = 1                                        !    OUTPUT:
      X = 2355.95843                               !
      WRITE (6,770) M,X                            !  56   2355.96
      WRITE (6,CHARVAR) M,M                         ! integer: 56
                                                   ! binary: 00111000
      WRITE (6,880) J,M                            !  1
```

```
770   FORMAT(I3, 2F10.2)
880   FORMAT(I<J+1>)
      END
```

Related information

- Chapter 10, "Input/Output formatting," on page 237
- "PRINT" on page 430
- "READ" on page 441
- "WRITE" on page 495

FUNCTION

Purpose

The **FUNCTION** statement is the first statement of a function subprogram.

Syntax

Notes:

1   IBM extension.

2   Fortran 2003
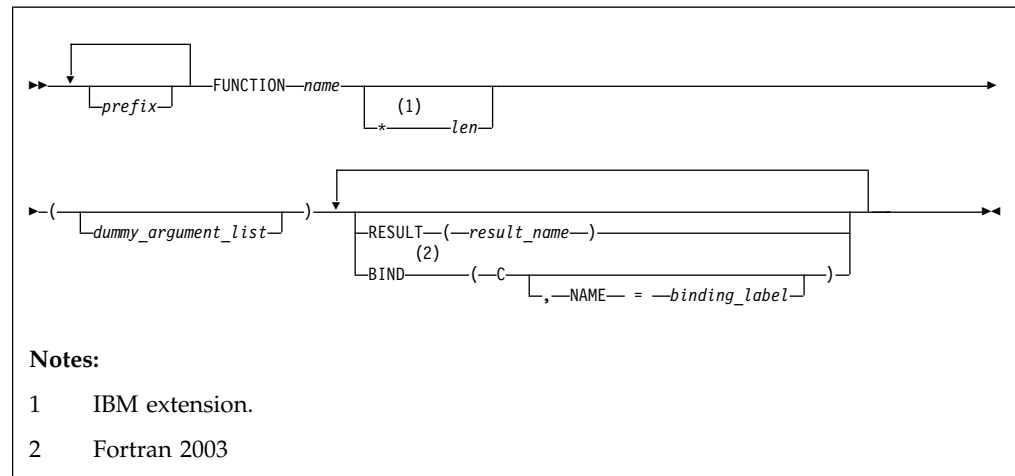
prefix is one of the following:

    declaration_type_spec
    ELEMENTAL
    F2008 **IMPURE** F2008
    F2008 **MODULE** F2008
    PURE
    RECURSIVE

declaration_type_spec
    specifies the type and type parameters of the function result. See "Type Declaration" on page 476 for details about *declaration_type_spec*.

name The name of the function subprogram.

IBM *len*
    An unsigned integer literal or a parenthesized scalar integer constant expression. The value of *len* specifies the length of the function's result variable. It can be included only when you specify the type in the **FUNCTION** statement. The type cannot be **DOUBLE PRECISION**, **DOUBLE COMPLEX**, **BYTE**, or a derived type. IBM

F2003 *binding_label*
> a scalar character constant expression. **F2003**

Rules

At most one of each kind of *prefix* can be specified. You cannot specify both the **RECURSIVE** and **ELEMENTAL** prefix specifiers. **F2008** You cannot specify both the **PURE** and **IMPURE** prefix specifiers. **F2008**

At most one **RESULT** clause and at most one **BIND** clause may appear. They can appear in any order.

The type and type parameters of the function result can be specified by either *declaration_type_spec* or by declaring the result variable in the declaration part of the function subprogram, but not by both. If they are not specified at all, the implicit typing rules are in effect. A length specifier cannot be specified by both *declaration_type_spec* and *len*.

If **RESULT** is specified, *result_name* becomes the function result variable. *name* must not be declared in any specification statement in the subprogram, although it can be referenced. *result_name* must not be the same as *name*. If **RESULT** is not specified, *name* becomes the function result variable.

F2003 The **BIND** keyword implicitly or explicitly defines a binding label by which a procedure is accessed from the C programming language. The result variable must be a scalar that is interoperable. A dummy argument cannot be zero-sized. A dummy argument for a procedure with the **BIND** attribute must have interoperable types and type parameters, and cannot have the **ALLOCATABLE** or **POINTER** attribute.

If the **FUNCTION** statement appears as part of an interface body that describes a dummy procedure, the **NAME=** specifier must not appear. An elemental procedure cannot have the **BIND** attribute. **F2003**

―――――――――――――――――――――― **End of** ――――――――――――――――――――――

If the result variable is an array or pointer, the **DIMENSION** or **POINTER** attributes, respectively, must be specified within the function body.

If the function result is a pointer, the shape of the result variable determines the shape of the value returned by the function. If the result variable is a pointer, the function must either associate a target with the pointer or define the association status of the pointer as disassociated.

If the result variable is not a pointer, the function must define its value.

If the name of an external function is of derived type, the derived type must be a sequence derived type if the type is not use-associated or host-associated.

The function result variable must not appear within a variable format expression, nor can it be specified in a **COMMON**, **DATA**, integer **POINTER**, or **EQUIVALENCE** statement, nor can it have the **PARAMETER**, **INTENT**, **OPTIONAL**, or **SAVE** attributes. The **STATIC** and **AUTOMATIC** attributes can be

specified only when the result variable is not an allocatable object, an array or a pointer, and is not of character or derived type.

The function result variable is associated with any entry procedure result variables. This is called entry association. The definition of any of these result variables becomes the definition of all the associated variables having that same type and type parameters, and is the value of the function regardless of the entry point.

If the function subprogram contains entry procedures, the result variables are not required to be of the same type unless the type is of character or derived type, or if the variables have the **ALLOCATABLE** or **POINTER** attribute, or if they are not scalars. 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 and type parameters redefines it later during execution of the subprogram.

`F2008`

You can specify the **MODULE** prefix specifier for the **FUNCTION** statement of a module subprogram or of a nonabstract interface body that is declared in the scoping unit of a module or submodule. See Example 2.

- When you specify the **MODULE** prefix specifier for the **FUNCTION** statement of a module subprogram, the module subprogram is a separate module procedure.
- When you specify the **MODULE** prefix specifier for the **FUNCTION** statement of a nonabstract interface body, the interface body is a module procedure interface body.

The **BIND** attribute with the **NAME=** specifier is not allowed on an internal procedure.

`F2008`

Example 1

```
RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
  INTEGER RES
  IF (N.EQ.0) THEN
    RES=1
  ELSE
    RES=N*FACTORIAL(N-1)
  END IF
END FUNCTION FACTORIAL

PROGRAM P
  INTERFACE OPERATOR (.PERMUTATION.)
    ELEMENTAL FUNCTION MYPERMUTATION(ARR1,ARR2)
      INTEGER :: MYPERMUTATION
      INTEGER, INTENT(IN) :: ARR1,ARR2
    END FUNCTION MYPERMUTATION
  END INTERFACE

  INTEGER PERMVEC(100,150),N(100,150),K(100,150)
  ...
  PERMVEC = N .PERMUTATION. K
  ...
END
```

Example 2 (Fortran 2008)

```
MODULE m
  ! The MODULE prefix specifier is specified for the FUNCTION
  ! statement of a module procedure interface body.
  INTERFACE
    INTEGER MODULE FUNCTION func(a, b)
      INTEGER :: a, b
    END FUNCTION
  END INTERFACE
END MODULE

SUBMODULE (m) n

  CONTAINS
    ! The MODULE prefix specifier is specified for the FUNCTION
    ! statement of a separate module procedure.
    INTEGER MODULE FUNCTION func(a, b) RESULT (res)
      INTEGER :: a, b
      res = a + b
    END FUNCTION
END SUBMODULE
```

Related information

- "Function and subroutine subprograms" on page 184
- "ENTRY" on page 358
- "BIND (Fortran 2003)" on page 298
- "Function reference" on page 186
- "Dummy arguments" on page 194
- "Statement Function" on page 461
- "Recursion" on page 208
- **-qrecur** option in the *XL Fortran Compiler Reference*
- "Pure procedures" on page 208
- "Elemental procedures" on page 211
- "Interoperability of procedures" on page 888
- "Modules" on page 177
- "Submodules (Fortran 2008)" on page 181
- "Separate module procedures (Fortran 2008)" on page 188

Recursion

The **RECURSIVE** keyword must be specified if, directly or indirectly:

- The function invokes itself
- The function invokes a function defined by an **ENTRY** statement in the same subprogram
- An entry procedure in the same subprogram invokes itself
- An entry procedure in the same subprogram invokes another entry procedure in the same subprogram
- An entry procedure in the same subprogram invokes the subprogram defined by the **FUNCTION** statement.

A function that directly invokes itself requires that both the **RECURSIVE** and **RESULT** keywords be specified. The presence of both keywords makes the procedure interface explicit within the subprogram.

If *name* is of type character, its length cannot be an asterisk if the function is recursive.

IBM If **RECURSIVE** is specified, the result variable has a default storage class of automatic.

You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the **FUNCTION** statement specifies either **RECURSIVE** or **RESULT**. **IBM**

Elemental procedures

For elemental procedures, the keyword **ELEMENTAL** must be specified. If the **ELEMENTAL** keyword is specified, the **RECURSIVE** keyword cannot be specified.

GO TO (assigned)

Purpose

The assigned **GO TO** statement transfers program control to an executable statement, whose statement label is designated in an **ASSIGN** statement.

Syntax

```
►►──GO TO──variable_name──────────────────────────────────────►◄
                          └─┬───┬─(──stmt_label_list──)─┘
                            └─,─┘
```
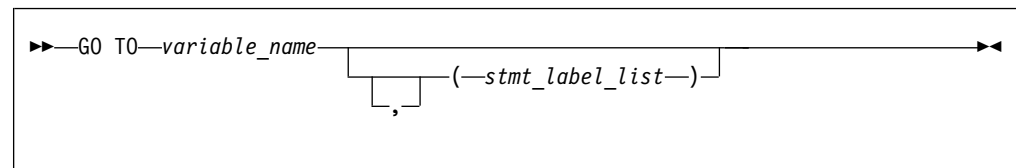
variable_name
> is a scalar variable name of type **INTEGER(4)** or **INTEGER(8)** that you have assigned a statement label to in an **ASSIGN** statement.

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

Rules

When 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 scoping unit as the assigned **GO TO** statement. If the integer variable is a dummy argument in a subprogram, you must assign it a statement label in the subprogram in order to use it in an assigned **GO TO** in that subprogram. 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.

The assigned **GO TO** cannot be the terminal statement of a **DO**, **F2008** **DO CONCURRENT** **F2008** , or **DO WHILE** construct.

The assigned **GO TO** statement has been deleted in Fortran 95.

Examples

```
      INTEGER RETURN_LABEL
         .
         .
         .
!  Simulate a call to a local procedure
      ASSIGN 100 TO RETURN_LABEL
      GOTO 9000
100   CONTINUE
         .
         .
         .
9000  CONTINUE
!  A "local" procedure
         .
         .
         .
      GOTO RETURN_LABEL
```

Related information

• "Statement labels" on page 9
• "Branching" on page 149
• "Deleted features" on page 1054

GO TO (computed)

Purpose

The computed **GO TO** statement transfers program control to one of possibly several executable statements.

Syntax
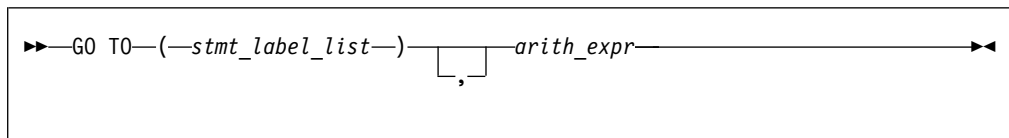
```
►►──GO TO──(──stmt_label_list──)──┬──────┬──arith_expr───────────────────►◄
                                  └──,──┘
```

stmt_label

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

arith_expr

is a scalar integer expression.

IBM It can also be real or complex. If the value of the expression is noninteger, XL Fortran converts it to **INTEGER(4)** before using it. **IBM**

Rules

When a computed **GO TO** statement is executed, the *arith_expr* is evaluated. The resulting value is used 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 four 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.

Examples

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

Related information
- "Statement labels" on page 9
- "Branching" on page 149

GO TO (unconditional)

Purpose

The unconditional **GO TO** statement transfers program control to a specified executable statement.

Syntax

```
►►──GO TO──stmt_label──────────────────────────────────────────────►◄
```

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

Rules

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

The unconditional **GO TO** statement cannot be the terminal statement of a **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** construct.

Examples

```
   REAL(8) :: X,Y
   GO TO 10
      ...
10 PRINT *, X,Y
   END
```

Related information
- "Statement labels" on page 9
- "Branching" on page 149

IF (arithmetic)

Purpose

The arithmetic **IF** statement transfers program control to one of three executable statements, depending on the evaluation of an arithmetic expression.

Syntax

```
►►──IF──(──arith_expr──)──stmt_label1──,──stmt_label2──,──stmt_label3──────────►◄
```

arith_expr
> is a scalar arithmetic expression of type integer or real

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

Rules

The arithmetic **IF** statement evaluates *arith_expr* and transfers 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.

Examples

```
      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
```

Related information
- "Branching" on page 149
- "Statement labels" on page 9

IF (block)

Purpose

The block **IF** statement is the first statement in an **IF** construct.

Syntax

```
►►──┬─────────────────────┬──IF─(─scalar_logical_expr─)─THEN──────────────►◄
    └─IF_construct_name─:─┘
```
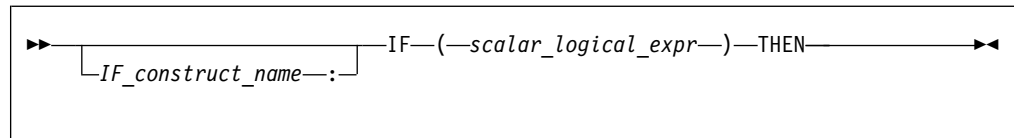
IF_construct_name
        Is a name that identifies the **IF** construct.

Rules

The block **IF** statement evaluates a logical expression and executes at most one of the blocks contained within the **IF** construct.

If the *IF_construct_name* is specified, it must appear on the **END IF** statement, and optionally on any **ELSE IF** or **ELSE** statements in the **IF** construct.

Examples

```
WHICHC: IF (CMD .EQ. 'RETRY') THEN
     IF (LIMIT .GT. FIVE) THEN          ! Nested IF constructs
        ...
         CALL STOP
     ELSE
         CALL RETRY
     END IF
ELSE IF (CMD .EQ. 'STOP') THEN WHICHC
     CALL STOP
ELSE IF (CMD .EQ. 'ABORT') THEN
     CALL ABORT
ELSE WHICHC
     GO TO 100
END IF WHICHC
```

Related information

IF (logical)

Purpose

The logical **IF** statement evaluates a logical expression and, if true, executes a specified statement.

Syntax

```
►►──IF─(─logical_expr─)─stmt──────────────────────────────────────────────►◄
```
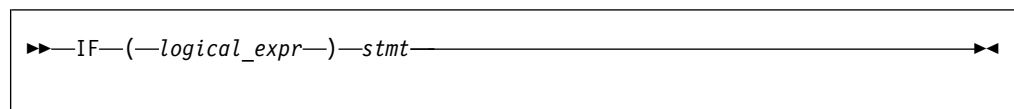
logical_expr
        is a scalar logical expression

stmt   is an unlabeled executable statement

Rules

When a logical **IF** statement is executed, the *logical_expr* is evaluated. If the value of *logical_expr* is true, *stmt* is executed. If the value of *logical_expr* is false, *stmt* does not execute and the **IF** statement has no effect (like a **CONTINUE** statement).

Execution of a function reference in *logical_expr* can change the values of variables that appear in *stmt*.

stmt cannot be a **SELECT CASE, CASE, END SELECT, DO,** `F2008` **DO CONCURRENT** `F2008` , **DO WHILE, END DO,** block **IF, ELSE IF, ELSE, END IF, END FORALL,** another logical **IF, ELSEWHERE, END WHERE, END, END FUNCTION, END SUBROUTINE** statement, **ASSOCIATE** construct statement, **FORALL** construct statement, or **WHERE** construct statement.

Examples

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

Related information

• Chapter 7, "Execution control," on page 135

IMPLICIT

Purpose

The **IMPLICIT** statement changes or confirms the default implicit typing or the default storage class for local entities or, with the form **IMPLICIT NONE** specified, voids the implicit type rules altogether.

Syntax

```
>>-IMPLICIT--+-NONE----------------------------------------------+-><
             |                                                   |
             |   .-,------------------------------------.        |
             |   V                                      |        |
             '-----+-declaration_type_spec--+--(--range_list--)--+-'
                   |          (1)           |
                   +-STATIC-----------------+
                   |          (2)           |
                   +-AUTOMATIC--------------+
                   |          (3)           |
                   '-UNDEFINED--------------'
```

Notes:

1   IBM extension.

2   IBM extension.

3   IBM extension.
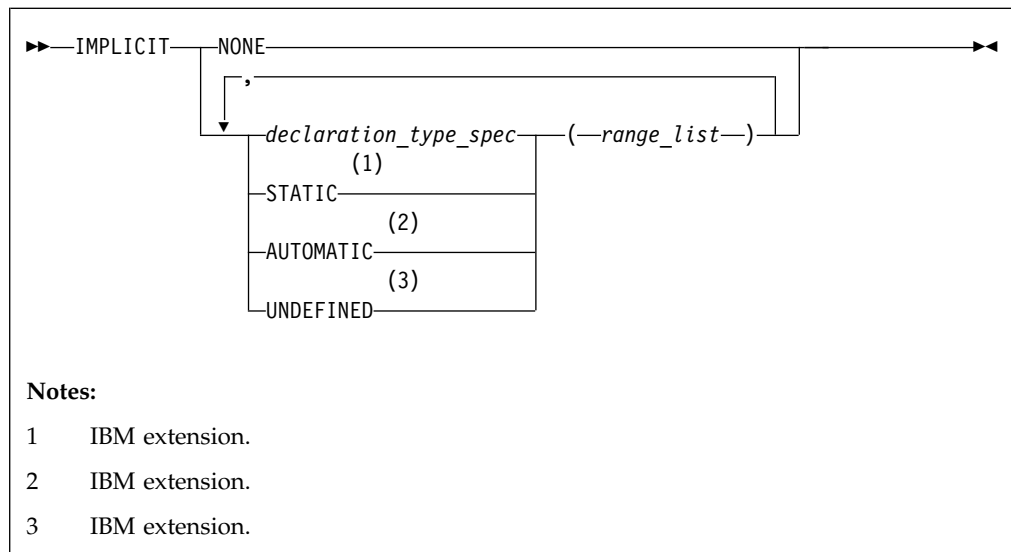
declaration_type_spec
> specifies a data type. See "Type Declaration" on page 476.

range   is either a single letter or range of letters. A range of letters has the form
> *letter₁-letter₂*, where *letter₁* is the first letter in the range and *letter₂*, which

follows *letter*$_1$ alphabetically, is the last letter in the range. Dollar sign ($) and underscore (_) are also permitted in a range. The underscore (_) follows the dollar sign ($), which follows the Z. Thus, the range Y - _ is the same as Y, Z, $, _.

Rules

Letter ranges cannot overlap; that is, no more than one type can be specified for a given letter.

In a given scoping unit, if a character has not been specified in an **IMPLICIT** statement, the implicit type for entities in a program unit or interface body is default integer for entities that begin with the characters I-N, and default real otherwise. The default for an internal or module procedure is the same as the implicit type used by the host scoping unit.

For any data entity name that begins with the character specified by *range_list*, and for which you do not explicitly specify a type, the type specified by the immediately preceding *declaration_type_spec* is provided. Note that implicit typing can be to a derived type that is inaccessible in the local scope if the derived type is accessible to the host scope.

F2008 The implicit typing rules of the host scoping unit also apply within a **BLOCK** construct. F2008

IBM A type specified in an **IMPLICIT** statement must not be a VECTOR type.

Deferred length type parameters cannot be specified in *declaration_type_spec*.

A character or a range of characters that you specify as **STATIC** or **AUTOMATIC** can also appear in an **IMPLICIT** statement for any data type. A letter in a *range_list* cannot have both *declaration_type_spec* and **UNDEFINED** specified for it in the scoping unit. Neither can both **STATIC** and **AUTOMATIC** be specified for the same letter. IBM

If you specify the form **IMPLICIT NONE** in a scoping unit, you must use type declaration statements to specify data types for names local to that scoping unit. You cannot refer to a name that does not have an explicitly defined data type; this lets you control all names that are inadvertently referenced. When **IMPLICIT NONE** is specified, you cannot specify any other **IMPLICIT** statement in the same scoping unit, except ones that contain **STATIC** or **AUTOMATIC**. You can compile your program with the **-qundef** compiler option to achieve the same effect as an **IMPLICIT NONE** statement appearing in each scoping unit where an **IMPLICIT** statement is allowed.

IBM **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 scoping unit that start with a specified character. The compiler issues a diagnostic message for each symbolic name local to the scoping unit that does not have an explicitly defined data type. IBM

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

IBM Using the **-qsave**/**-qnosave** compiler option modifies the predefined conventions for storage class:

-qsave compiler option	makes the predefined convention	IMPLICIT STATIC(a - _)
-qnosave compiler option	makes the predefined convention	IMPLICIT AUTOMATIC(a - _)

Even if you specified the **-qmixed** compiler option, the range list items are not case sensitive. For example, with **-qmixed** specified, IMPLICIT INTEGER(A) affects the implicit typing of data objects that begin with A as well as those that begin with a.
IBM

Examples

```
    IMPLICIT INTEGER (B), COMPLEX (D, K-M), REAL (R-Z,A)
! This IMPLICIT statement establishes the following
! implicit typing:
!
!        A: real
!        B: integer
!        C: real
!        D: complex
!   E to H: real
!     I, J: integer
! K, L, M: complex
!        N: integer
!   O to Z: real
!        $: real
!        _: real
```

Related information

- "Determining Type" on page 18 for a discussion of the implicit rules
- "Storage classes for variables (IBM extension)" on page 28
- **-qundef** option in the *XL Fortran Compiler Reference*
- **-qsave** option in the *XL Fortran Compiler Reference*

IMPORT (Fortran 2003)

Purpose

Named entities from the host scoping unit are not accessible in an interface body that is not a module procedure interface body. The **IMPORT** statement makes those entities accessible in such interface body by host association. The **IMPORT** statement is not valid in a module procedure interface body.

Syntax

```
►►──IMPORT─────────────────────────────────────────────►◄
              └──┬────┬──import_name_list──┘
                 └─::─┘
```
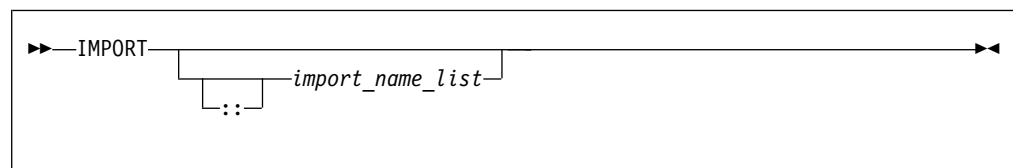
import_name_list
        is a list of named entities that are accessible in the host scoping unit

Rules

The **IMPORT** statement is allowed only in an interface body with the exception of the module procedure interface body. Each of the specified names must be explicitly declared before the interface body.

The entities in the import name list are imported into the current scoping unit and are accessible by host association. If no names are specified, all of the accessible named entities in the host scoping unit are imported.

The names of imported entities must not appear in any context that would cause the host entity to be inaccessible.

Examples

```
use, intrinsic :: ISO_C_BINDING
interface
  subroutine process_buffer(buffer, n_bytes), bind(C,NAME="ProcessBuffer")
    IMPORT :: C_PTR, C_INT
    type (C_PTR), value :: buffer
    integer (C_INT), value :: n_bytes
  end subroutine process_buffer
end interface
......
```

Related information
- "INTERFACE" on page 405
- "Host association" on page 156
- "Interface concepts" on page 162

INQUIRE

Purpose

The **INQUIRE** statement obtains information about the properties of a named file or the connection to a particular unit.

There are three forms of the **INQUIRE** statement:
- Inquire by file, which requires the **FILE=** specifier.
- Inquire by output list, which requires the **IOLENGTH=** specifier
- Inquire by unit, which requires the **UNIT=** specifier.

Syntax

```
►►──INQUIRE──┬─(─inquiry_list─)─────────────────────┬──►◄
             └─(─IOLENGTH─=─iol─)─output_item_list───┘
```
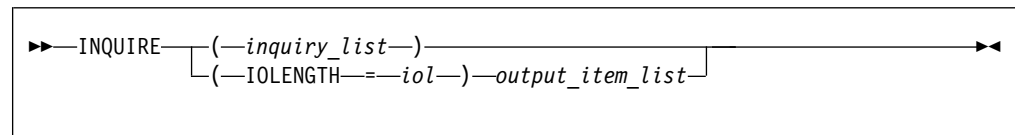
iol   indicates the number of bytes of data that would result from the use of the output list in an unformatted output statement. *iol* is a scalar integer variable.

output_item
        See the **PRINT** or **WRITE** statement

inquiry_list

is a list of inquiry specifiers for the inquire-by-file and inquire-by-unit forms of the **INQUIRE** statement. The inquire-by-file form cannot contain a unit specifier, and the inquire-by-unit form cannot contain a file specifier. No specifier can appear more than once in any **INQUIRE** statement. The inquiry specifiers are:

[UNIT=] *u*

is a unit specifier. It specifies the unit about which the inquire-by-unit form of the 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. It is one of the following:

- An integer expression whose value is in the range 1 through 2147483647
- F2008 A NEWUNIT value F2008

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

ACCESS= *char_var*

indicates whether the file is connected for direct access, sequential access, F2003 or stream access. F2003 *char_var* is a scalar character variable that is assigned the value **SEQUENTIAL** if the file is connected for sequential access. The value assigned is **DIRECT** if the file is connected for direct access. F2003 The value assigned is **STREAM** if the file is connected for stream access. F2003 If there is no connection, *char_var* is assigned the value **UNDEFINED**.

ACTION= *act*

indicates if the file is connected for read and/or write access. *act* is a scalar character variable that is assigned the value **READ** if the file is connected for input only, **WRITE** if the file is connected for output only, **READWRITE** if the file is connected for both input and output, and **UNDEFINED** if there is no connection.

ASYNCH= *char_variable* **(IBM extension)**

indicates whether the unit is connected for asynchronous access.

char_variable is a character variable that is assigned the value:

- **YES** if the unit is connected for both synchronous and asynchronous access;
- **NO** if the unit is connected for synchronous access only; or
- **UNDEFINED** if the unit is not connected.

ASYNCHRONOUS= *char_var* **(Fortran 2003)**

indicates whether the file is connected and asynchronous I/O on the unit is allowed.

char_var is a character variable that is assigned the value:

- **YES** if the file is connected and asynchronous I/O on the unit is allowed;
- **NO** if the file is connected and asynchronous I/O on the unit is not allowed; or
- **UNDEFINED** if the file is not connected.

An IBM **ASYNCH=** IBM specifier and an F2003 **ASYNCHRONOUS=** specifier F2003 should not appear on the same **INQUIRE** statement, the second one is ignored.

BLANK= *char_var*

indicates the default treatment of blanks for a file connected for formatted input/output. *char_var* is a scalar character variable that is assigned the value **NULL** if all blanks in numeric input fields are ignored, or the value **ZERO** if all nonleading blanks are interpreted as zeros. If there is no connection, or if the connection is not for formatted input/output, *char_var* is assigned the value **UNDEFINED**.

CONVERT= *char_var*

indicates the byte order for I/O operations on the file. *char_var* is a scalar character variable that is assigned the value **BIG_ENDIAN** if big-endian byte order is used for I/O operations on the file. Otherwise, *char_var* is assigned the value **NATIVE**.

DECIMAL= *char_var* **(Fortran 2003)**

char_var is a scalar character variable which is assigned a value of either **POINT**, or **COMMA**, corresponding to the decimal edit mode in effect for a formatted input/output connection. If there is no connection, or if the connection is not for formatted input/output, *char_var* is assigned the value **UNDEFINED**.

DELIM= *del*

indicates the form, if any, that is used to delimit character data that is written by list-directed or namelist formatting. *del* is a scalar character variable that is assigned the value **APOSTROPHE** if apostrophes are used to delimit data, **QUOTE** if quotation marks are used to delimit data, **NONE** if neither apostrophes nor quotation marks are used to delimit data, and **UNDEFINED** if there is no file connection or no connection to formatted data.

DIRECT= *dir*

indicates if the file is connected for direct access. *dir* is a scalar character variable 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 access cannot be determined.

ENCODING= *char_variable* **(Fortran 2003)**

indicates the encoding form of the file. *char_variable* is a character variable that is assigned the value **DEFAULT** if the encoding form of the file is ASCII, **UNDEFINED** if the I/O connection is unformatted, and **UNKNOWN** if there is no file connection.

ERR= *stmt_label*

is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

EXIST= *ex*

indicates if a file or unit exists. *ex* is an integer variable that is assigned the value `true` or `false`. For the inquire-by-file form of the statement, the value `true` is assigned if the file specified by the **FILE=** specifier exists. The value `false` is assigned if the file does not exist. For the inquire-by-unit form of the statement, the value `true` is assigned if the unit specified by **UNIT=** exists. The value `false` is assigned if it is an invalid unit.

FILE= *char_expr*

is a file specifier. It specifies the name of the file about which the inquire-by-file form of the statement is inquiring. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is

a valid Linux operating system file name. The named file does not have to exist, nor does it have to be associated with a unit.

IBM

Note: A valid Linux operating system file name must have a full path name of total length ≤ 1023 characters, with each file name ≤ 255 characters long (though the full path name need not be specified).
IBM

FORM= *char_var*
indicates whether the file is connected for formatted or unformatted input/output. *char_var* is a scalar default character variable that is assigned the value **FORMATTED** if the file is connected for formatted input/output. The value assigned is **UNFORMATTED** if the file is connected for unformatted input/output. If there is no connection, *char_var* is assigned the value **UNDEFINED**.

FORMATTED= *fmt*
indicates if the file can be connected for formatted input/output. *fmt* is a scalar character variable 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 formatting cannot be determined.

ID= *scalar_int_expr* **(Fortran 2003)**
is a specifier that identifies a pending data transfer operation for a specified unit. *scalar_int_expr* is a scalar default character variable.

If an **ID=** specifier appears and the specified data transfer operation is complete, then the variable specified in the **PENDING=** specifier is assigned the value `false` and the **INQUIRE** statement performs the wait operation for the specified data transfer.

If there is no **ID=** specifier and all data transfer operations for the specified unit are complete, then the variable specified in the **PENDING=** specifier is assigned the value `false` and the **INQUIRE** statement performs wait operations for all previously pending data transfers for the specified unit.

PENDING= specifier will be assigned the value `true` in all other cases and no wait operations will be performed. Previously pending data transfers will remain pending after the execution of the **INQUIRE** statement.

IOMSG= *iomsg_variable* **(Fortran 2003)**
is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:
- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*
is an input/output status specifier that specifies the status of the input/output operation. *ios* is an integer variable. When the input/output statement containing this specifier is finished executing, *ios* is defined with:
- A zero value if no error condition occurs
- A positive value if an error occurs.

Coding the **IOSTAT=** specifier suppresses error messages.

NAME= *fn*

indicates the name of the file. *fn* is a scalar character variable that is assigned the name of the file to which the unit is connected.

NAMED= *nmd*

indicates if the file has a name. *nmd* is an integer variable that is assigned the value `true` if the file has a name. The value assigned is `false` if the file does not have a name.

NEXTREC= *nr*

indicates where the next record can be read or written on a file connected for direct access. *nr* is an integer variable 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.

IBM Because record numbers can be greater than 2**31-1, you may choose to make the scalar variable specified with the **NEXTREC=** specifier of type **INTEGER(8)**. This could be accomplished in many ways, two examples include:

- Explicitly declaring *nr* as **INTEGER(8)**.
- Changing the default kind of integers with the **-qintsize=8** compiler option.

 ◀IBM

NUMBER= *num*

indicates the external unit identifier currently associated with the file. *num* is an integer variable 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* is assigned the value -1.

OPENED= *od*

indicates if a file or unit is connected. *od* is an integer variable that is assigned the value `true` or `false`. For the inquire-by-file form of the statement, the value `true` is assigned if the file specified by **FILE=** *char_var* is connected to a unit. The value `false` is assigned if the file is not connected to a unit. For the inquire-by-unit form of the statement, the value `true` is assigned if the unit specified by **UNIT=** is connected to a file. The value `false` is assigned if the unit is not connected to a file. For preconnected files that have not been closed, the value is `true` both before and after the first input/output operation.

PAD= *pd*

indicates the current **PAD** mode of the connection. *pd* is a scalar character variable with the default value **YES**. *pd* is assigned the value **NO** if the connection of the file specifies **PAD=NO**.

PAD= returns **UNDEFINED** when there is no connection, or when the connection is for unformatted I/O. If you compile your application with **-qxlf90=oldpad**, **PAD=** returns **YES** in these cases.

PENDING=*scalar_default_logical_variable* **(Fortran 2003)**

indicates whether or not previously pending asynchronous data transfers are complete. A data transfer operation is previously pending if it is

pending at the begining of execution of the **INQUIRE** statement.
scalar_default_logical_variable is an integer variable that is assigned the value
`true` or `false`.

If an **ID=** specifier appears and the specified data transfer operation is
complete, then the variable specified in the **PENDING=** specifier is
assigned the value `false` and the **INQUIRE** statement performs the wait
operation for the specified data transfer.

If there is no **ID=** specifier and all data transfer operations for the specified
unit are complete, then the variable specified in the **PENDING=** specifier
is assigned the value `false` and the **INQUIRE** statement performs wait
operations for all previously pending data transfers for the specified unit.

PENDING= specifier will be assigned the value `true` in all other cases and
no wait operations will be performed. Previously pending data transfers
will remain pending after the execution of the **INQUIRE** statement.

POS=*integer_var* **(Fortran 2003)**
> *integer_var* is an integer variable that indicates the value of the file position
> for a file connected for stream access. *integer_var* is assigned the number of
> the file storage unit immediately following the current position of a file
> connected for stream access. If the file is positioned at its terminal position,
> *integer_var* is assigned a value one greater than the highest-numbered
> storage unit in the file. *integer_var* becomes undefined if the file is not
> connected for stream access or if the position of the file can not be
> determined because of previous error conditions.

POSITION= *pos*
> indicates the position of the file. *pos* is a scalar character variable that is
> assigned the value **REWIND** if the file is connected by an **OPEN** statement
> for positioning at its initial point, **APPEND** if the file is connected for
> positioning before its endfile record or at its terminal point, **ASIS** if the file
> is connected without changing its position, or **UNDEFINED** if there is no
> connection or if the file is connected for direct access.

> If the file has been repositioned to its initial point since it was opened, *pos*
> is assigned the value **REWIND**. If the file has been repositioned just before
> its endfile record since it was opened (or, if there is no endfile record, at its
> terminal point), *pos* is assigned the value **APPEND**. If both of the above
> are true and the file is empty, *pos* is assigned the value **APPEND**. If the file
> is positioned after the endfile record, *pos* is assigned the value **ASIS**.

READ= *rd*
> indicates if the file can be read. *rd* is a scalar character variable that is
> assigned the value **YES** if the file can be read, **NO** if the file cannot be
> read, and **UNKNOWN** if it cannot be determined if the file can be read.

READWRITE= *rw*
> indicates if the file can be both read from and written to. *rw* is a scalar
> character variable that is assigned the value **YES** if the file can be both
> read from and written to, **NO** if the file cannot be both read from and
> written to, and **UNKNOWN** if it cannot be determined if the file can be
> both read from and written to.

RECL= *rcl*

> indicates the value of the record length of a file connected for direct access,
> or the value of the maximum record length of a file connected for
> sequential access.

rcl is an integer variable that is assigned the value of the record length.

If the file is connected for formatted input/output, the length is the number of characters for all records that contain character data. If the file is connected for unformatted input/output, the length is the number of bytes of data. If there is no connection, *rcl* becomes undefined.

F2003 If the file is connected for stream access, *rcl* becomes undefined. F2003

ROUND= *char_var* **(Fortran 2003)**

assigns the value **UP**, **DOWN**, **ZERO**, **PROCESSOR_DEPENDENT**, **NEAREST** or **COMPATIBLE**, (whichever is the rounding mode for the current connection) to *char_var*. If there is no connection or the input is not formatted, the returned value is **UNDEFINED**. *char_var* is a character variable.

The rounding mode helps specify how decimal numbers are converted to an internal representation, (that is, in binary) from a character representation and vice versa during formatted input and output. The rounding modes have the following functions:

- In the **UP** rounding mode the value from the conversion is the smallest value that is greater than or equal to the original value.
- In the **DOWN** rounding mode the value from the conversion is the greatest value that is smaller than or equal to the original value.
- In the **ZERO** rounding mode the value from the conversion is the closest value to the original value, and not greater in magnitude.
- In the **NEAREST** rounding mode the value from the conversion is the closer of the two nearest representable values. If both values are equally close then the even value will be chosen. In IEEE rounding conversions, NEAREST corresponds to the ieee_nearest rounding mode as specified by the IEEE standard.
- In the **COMPATIBLE** rounding mode the value from the conversion is the closest of the two nearest representable values, or the value further away from zero if halfway between.
- In the **PROCESSOR_DEFINED** rounding mode the value from the conversion is processor dependent and may correspond to the other modes. In XL Fortran, the **PROCESSOR_DEFINED** rounding mode will be the rounding mode you choose in the floating-point control register. If you do not set the floating-point control register explicitly, the default rounding mode is **NEAREST**.

SEQUENTIAL= *seq*

indicates if the file is connected for sequential access. *seq* is a scalar character variable 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 access cannot be determined.

SIGN= *char_var* **(Fortran 2003)**

indicates the sign mode in effect for a connection for formatted input/output. If *char_var* is assigned the value **PLUS**, the processor shall produce a plus sign in any position that normally contains an optional plus sign and suppresses plus signs in these positions if *char_var* is assigned the value **SUPPRESS**. *char_var* can also be assigned the value **PROCESSOR_DEFINED** which is the default sign mode and acts the same as **SUPPRESS**. If there is no connection, or if the connection is not for formatted input/output, *char_var* is assigned the value **UNDEFINED**.

SIZE=*filesize*
> *filesize* is an integer variable that is assigned the file size in bytes.

STREAM=*strm* **(Fortran 2003)**
> is a scalar default character variable that indicates whether the file is connected for stream access. *strm* is assigned the value **YES** if the file can be accessed using stream access, the value **NO** if the file cannot be accessed using stream access, or the value **UNKNOWN** if access cannot be determined.

TRANSFER= *char_variable* **(IBM extension)**
> is an asynchronous I/O specifier that indicates whether synchronous and/or asynchronous data transfer are permissible transfer methods for the file.
>
> *char_variable* is a scalar character variable. If *char_variable* is assigned the value **BOTH**, then both synchronous and asynchronous data transfer are permitted. If *char_variable* is assigned the value **SYNCH**, then only synchronous data transfer is permitted. If *char_variable* is assigned the value **UNKNOWN**, then the processor is unable to determine the permissible transfer methods for this file.

UNFORMATTED= *unf*
> indicates if the file can be connected for unformatted input/output. *fmt* is a scalar character variable 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 formatting cannot be determined.

WRITE= *wrt*
> indicates if the file can be written to. *wrt* is a scalar character variable that is assigned the value **YES** if the file can be written to, **NO** if the file cannot be written to, and **UNKNOWN** if it cannot be determined if the file can be written to.

Rules

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.

IBM extension

If the unit or file is connected, the values returned for the **ACCESS=**, **SEQUENTIAL=**, **STREAM=**, **DIRECT=**, **ACTION=**, **READ=**, **WRITE=**, **READWRITE=**, **FORM=**, **FORMATTED=**, **UNFORMATTED=**, **BLANK=**, **DELIM=**, **PAD=**, **RECL=**, **POSITION=**, **NEXTREC=**, **NUMBER=**, **NAME=**, **NAMED=**, **DECIMAL=**, **ROUND=** and **SIGN=** specifiers are properties of the connection, and not of that file. Note that the **EXIST=** and **OPENED=** specifiers return `true` in these situations.

If a unit or file is not connected or does not exist, the **ACCESS=**, **ACTION=**, **FORM=**, **BLANK=**, **DELIM=**, **POSITION=** specifiers return the value **UNDEFINED**, the **DIRECT=**, **SEQUENTIAL=**, **STREAM=**, **FORMATTED=**, **UNFORMATTED=**, **READ=**, **WRITE=** and **READWRITE=** specifiers return the value **UNKNOWN**, the **RECL=** and **NEXTREC=** specifier variables are not defined, the **PAD=** specifier returns the value **YES**, and the **OPENED** specifier returns the value `false`. The value returned by the **SIZE=** specifier is -1.

If a unit or file does not exist, the **EXIST=** and **NAMED=** specifiers return the value `false`, the **NUMBER=** specifier returns the value -1, and the **NAME=** specifier variable is not defined.

If a unit or file exists but is not connected, the **EXIST=** specifier returns the value `true`. For the inquire-by-unit form of the statement, the **NAMED=** specifier returns the value `false`, the **NUMBER=** specifier returns the unit number, and the **NAME=** specifier variable is undefined. For the inquire-by-file form of the statement, the **NAMED=** specifier returns the value `true`, the **NUMBER=** specifier returns -1, and the **NAME=** specifier returns the file name.

──────────── **End of IBM extension** ────────────

The same variable name must not be specified for more than one specifier in the same **INQUIRE** statement, and must not be associated with any other variable in the list of specifiers.

Examples

```
SUBROUTINE SUB(N)
  CHARACTER(N) A(5)
  INQUIRE (IOLENGTH=IOL) A(1)  ! Inquire by output list
  OPEN (7,RECL=IOL)
   .
   .
   .
END SUBROUTINE
```

Related information

- "Conditions and IOSTAT values" on page 226
- Chapter 9, "XL Fortran Input/Output," on page 215

INTEGER

Purpose

An **INTEGER** type declaration statement specifies the length and attributes of objects and functions of type integer. Initial values can be assigned to objects.

Syntax

```
►►──INTEGER──┬────────────────┬──┬──────────────────────┬──entity_decl_list──►◄
             └─kind_selector──┘  ├─::───────────────────┤
                                 └─,──attr_spec_list──::─┘
```

where:

attr_spec
       is any of the following:

ALLOCATABLE [1]	**INTRINSIC**	**PUBLIC**
ASYNCHRONOUS	**OPTIONAL**	**SAVE**
AUTOMATIC [2]	**PARAMETER**	**STATIC** [2]
BIND [1]	**POINTER**	**TARGET**

DIMENSION (*array_spec*)	PRIVATE	VALUE **1**
EXTERNAL	PROTECTED **1**	VOLATILE
INTENT (*intent_spec*)		
Note: **1** Fortran 2003 **2** IBM extension		

kind_selector

Notes:

1 IBM extension.

> **IBM** specifies the length of integer entities: 1, 2, 4 or 8. *int_literal_constant* cannot specify a kind type parameter. **IBM**

attr_spec
> For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
> is a list of dimension bounds

entity_decl

```
►►──a──┬─────────────────────────────┬──┬──────────────────┬────────────►
       │           (1)        (2)    │  └─(──array_spec──)──┘
       └──* ─────────len──────────┘
       │                     (3)          (4)                │
       └──(──array_spec──)── * ─────────len──────────────────┘

►──┬──────────────────────────────────────────┬────────────────────────►◄
   │                                    (5)    │
   ├──/──initial_value_list──/─────────────────┤
   ├──= ──constant_expr────────────────────────┤
   └──=> ──NULL()──────────────────────────────┘
```

Notes:

1   IBM extension.

2   IBM extension.

3   IBM extension.

4   IBM extension.

5   IBM extension.

> *a*
>   is an object name or function name. *array_spec* cannot be specified for a function name with an implicit interface.
>
> **IBM** *len*
>   overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications. **IBM**
>
> **IBM** *initial_value*
>   provides an initial value for the entity specified by the immediately preceding name **IBM**
>
> *constant_expr*
>   provides a constant expression for the entity specified by the immediately preceding name
>
> => **NULL()**
>   provides the initial value for the pointer object

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit ▶IBM◀ or if it appears in a named common block in a module or submodule. ◀IBM

You can initialize pointers using **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined.

If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the ▶F2003◀ **ALLOCATABLE** or ◀F2003 **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

IBM If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. IBM

Examples

```
MODULE INT
  INTEGER, DIMENSION(3) :: A,B,C
  INTEGER :: X=234,Y=678
END MODULE INT
```

Related information

- "Integer" on page 37
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

INTENT

Purpose

The **INTENT** attribute specifies the intended use of dummy arguments.

Syntax

```
►►──INTENT──(──┬──IN───┬──)──┬────┬──dummy_arg_name_list────────►◄
               ├─OUT──┤      └─::─┘
               └─INOUT─┘
```
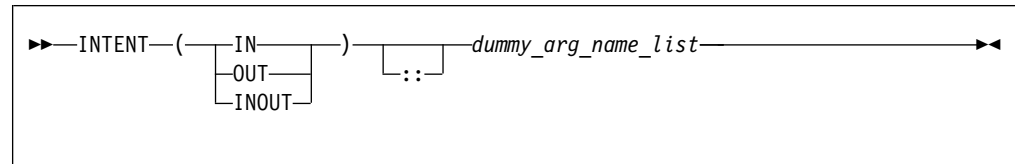
dummy_arg_name
    is the name of a dummy argument, which cannot be a dummy procedure

Rules

If you specify a nonpointer, nonallocatable dummy argument, the **INTENT** attribute will have the following characteristics:

- **INTENT(IN)** specifies that the dummy argument must not be redefined or become undefined during the execution of the subprogram.
- **INTENT(OUT)** specifies that the dummy argument must be defined before it is referenced within the subprogram. Such a dummy argument might not become undefined on invocation of the subprogram.
- **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

If you specify a pointer dummy argument, the **INTENT** attribute will have the following characteristics:

- **INTENT(IN)** specifies that during the execution of the procedure, the association status of the pointer dummy argument cannot be changed, except if the target of the pointer is deallocated. If the target of the pointer is deallocated, the association status of the pointer dummy argument becomes undefined.

You cannot use an **INTENT(IN)** pointer dummy argument as a pointer object in a pointer assignment statement. You cannot allocate, deallocate, or nullify an **INTENT(IN)** pointer dummy argument

You cannot specify an **INTENT(IN)** pointer dummy argument as an actual argument to a procedure if the associated dummy argument is a pointer with **INTENT(OUT)** or **INTENT(INOUT)** attribute.

- **INTENT(OUT)** specifies that at the execution of the procedure, the association status of the pointer dummy argument is undefined
- **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

If you specify an allocatable dummy argument, the **INTENT** attribute will have the following characteristics:

- **INTENT(IN)** specifies that during the execution of the procedure, the allocation status of the dummy argument cannot be changed, and it must not be redefined or become undefined.
- **INTENT(OUT)** specifies that at the execution of the procedure, if the associated actual argument is allocated it will be deallocated.
- **INTENT(INOUT)** specifies that the dummy argument can both receive and return data to the invoking subprogram.

If a dummy argument with **INTENT(OUT)** is a derived type with default initialization, the dummy argument cannot be an assumed-size array.

If you do not specify the **INTENT** attribute for a pointer or allocatable dummy argument, its use is subject to the limitations and restrictions of the associated actual argument.

An actual argument that becomes associated with a dummy argument with an intent of **OUT** or **INOUT** must be definable. Hence, a dummy argument with an intent of **IN**, or an actual argument that is a constant, a subobject of a constant, or an expression, cannot be passed as an actual argument to a subprogram expecting an argument with an intent of **OUT** or **INOUT**.

An actual argument that is an array section with a vector subscript cannot be associated with a dummy array that is defined or redefined (that is, with an intent of **OUT** or **INOUT**).

Table 44. Attributes compatible with the INTENT attribute

ALLOCATABLE **1**	CONTIGUOUS **2**	TARGET
ASYNCHRONOUS	OPTIONAL	VALUE **1**
DIMENSION	POINTER	VOLATILE
Note: **1** Fortran 2003 **2** Fortran 2008		

You must not specify the **VALUE** attribute for a dummy argument with an intent of **OUT** or **INOUT**

IBM The **%VAL** built-in function, used for interlanguage calls, can only be used for an actual argument that corresponds to a dummy argument with an intent of **IN**, or has no intent specified. This constraint does not apply to the **%REF** built-in function. **IBM**

Examples

```
PROGRAM MAIN
  DATA R,S /12.34,56.78/
  CALL SUB(R+S,R,S)
END PROGRAM

SUBROUTINE SUB (A,B,C)
  INTENT(IN) A
  INTENT(OUT) B
  INTENT(INOUT) C
  C=C+A+ABS(A)              ! Valid references to A and C
                           ! Valid redefinition of C
  B=C**2                   ! Valid redefinition of B
END SUBROUTINE
```

Related information

- "Intent of dummy arguments" on page 197
- "Argument association" on page 195
- "%VAL and %REF (IBM extension)" on page 196, for details on interlanguage calls
- "Dummy arguments" on page 194

INTERFACE

Purpose

The **INTERFACE** statement is the first statement of an interface block, which can specify an explicit interface for an external or dummy procedure.

Syntax
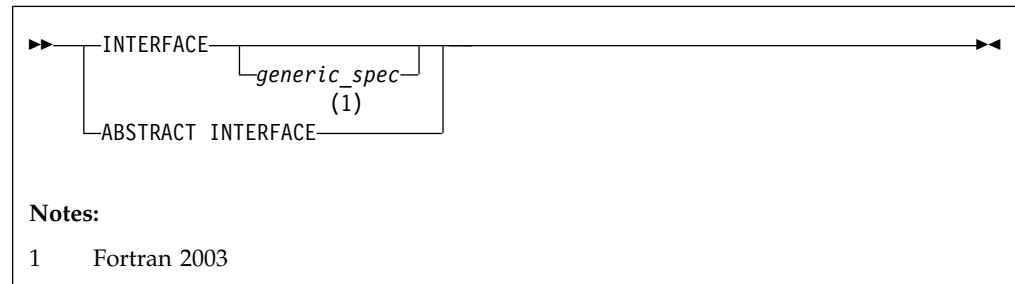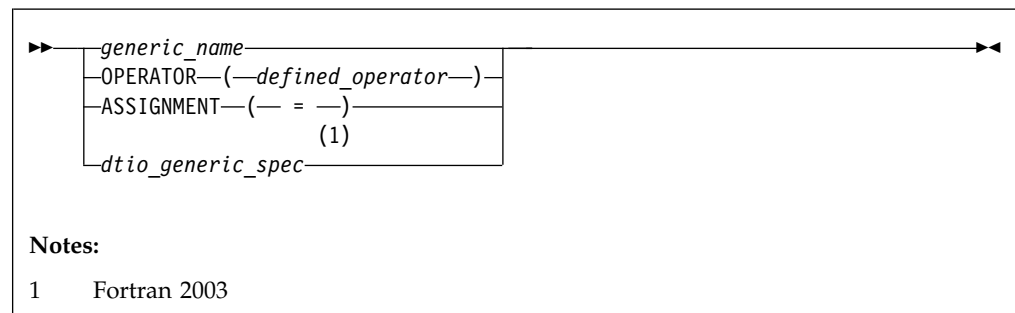
```
▶▶──┬─INTERFACE─────────────────────┬────────────────────────────▶◀
    │         └─generic_spec─┘       │
    │                  (1)           │
    └─ABSTRACT INTERFACE─────────────┘
```

Notes:

1   Fortran 2003

generic_spec
        is

```
▶▶──┬─generic_name──────────────────────┬──────────────────────▶◀
    ├─OPERATOR──(─defined_operator─)─────┤
    ├─ASSIGNMENT──(─ = ─)────────────────┤
    │              (1)                   │
    └─dtio_generic_spec──────────────────┘
```

Notes:

1   Fortran 2003

defined_operator

> is a defined unary operator, defined binary operator, or extended intrinsic operator

dtio_generic_spec

```
►►─┬─READ──(──FORMATTED──)───┬──────────────────────────────────────────►◄
   ├─READ──(──UNFORMATTED──)─┤
   ├─WRITE──(──FORMATTED──)──┤
   └─WRITE──(──UNFORMATTED──)┘
```
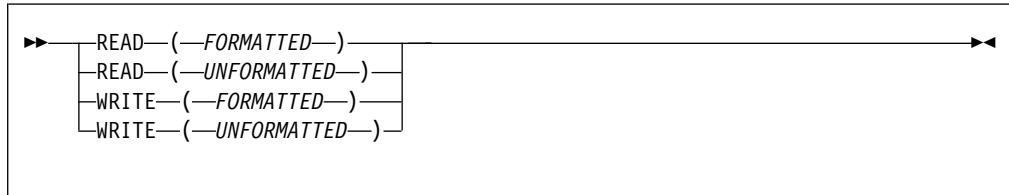
Rules

If *generic_spec* is present, the interface block is generic. If *generic_spec* and **ABSTRACT** is absent, the interface block is specific. An interface block introduced by **ABSTRACT INTERFACE** is an abstract interface block. *generic_name* specifies a single name to reference all procedures in the interface block. At most, one specific procedure is invoked each time there is a procedure reference with a generic name.

An interface body in a generic or specific interface block specifies the **EXTERNAL** attribute and an explicit specific interface for an external procedure or a dummy procedure. If the name of the declared procedure is that of a dummy argument in the subprogram containing the interface body, the procedure is a dummy procedure; otherwise, it is an external procedure.

If a *generic_spec* appears in an **INTERFACE** statement, it must match the *generic_spec* in the corresponding **END INTERFACE** statement.

If the *generic_spec* in an **INTERFACE** statement is a *generic_name*, the *generic_spec* of the corresponding **END INTERFACE** statement must be the same *generic_name*.

An **INTERFACE** statement without a *generic_spec* can match any **END INTERFACE** statement, with or without a *generic_spec*.

A specific procedure must not have more than one explicit interface in a given scoping unit.

You can always reference a procedure through its specific interface, if accessible. If a generic interface exists for a procedure, the procedure can also be referenced through the generic interface.

If *generic_spec* is **OPERATOR**(*defined_operator*), the interface block can define a defined operator or extend an intrinsic operator.

If *generic_spec* is **ASSIGNMENT**(=), the interface block can extend intrinsic assignment.

F2003 If *generic_spec* is *dtio_generic_spec*, the interface block defines derived type input/output procedures. User-defined derived type input/output procedures allow your application to override the default handling of derived type objects and

values in data transfer input/output statements. The subroutines in this interface
block must have interfaces described in "User-defined derived-type Input/Output
procedure interfaces (Fortran 2003)" on page 222. `F2003`

Examples

```
INTERFACE                              ! Nongeneric interface block
  FUNCTION VOL(RDS,HGT)
    REAL VOL, RDS, HGT
  END FUNCTION VOL
  FUNCTION AREA (RDS)
    REAL AREA, RDS
  END FUNCTION AREA
END INTERFACE

INTERFACE OPERATOR (.DETERMINANT.)    ! Defined operator interface
  FUNCTION DETERMINANT(X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE

INTERFACE ASSIGNMENT(=)                ! Defined assignment interface
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN)  :: B(:)
  END SUBROUTINE
END INTERFACE
```

Related information

- "Explicit interface" on page 164
- "Extended intrinsic and defined operations" on page 113
- "Defined operators" on page 170
- "Defined assignment" on page 172
- "User-defined derived-type Input/Output procedure interfaces (Fortran 2003)" on page 222
- "FUNCTION" on page 379
- "SUBROUTINE" on page 468
- "PROCEDURE" on page 433
- "Procedure references" on page 186
- "Unambiguous generic procedure references" on page 168, for details about the rules on how any two procedures with the same generic name must differ

INTRINSIC

Purpose

The **INTRINSIC** attribute identifies a name as an intrinsic procedure and allows
you to use specific names of intrinsic procedures as actual arguments.

Syntax

```
►►──INTRINSIC─┬────┬──name_list──────────────────────────────────────►◄
              └─::─┘
```
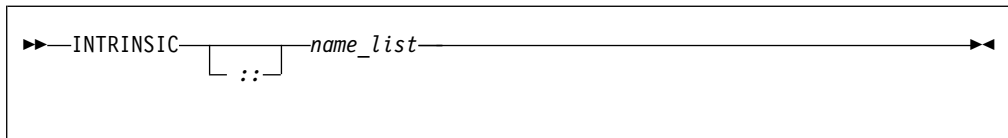
name   is the name of an intrinsic procedure

Rules

If you use a specific intrinsic procedure name as an actual argument in a scoping unit, it must have the **INTRINSIC** attribute. Generic names can have the **INTRINSIC** attribute, but you cannot pass them as arguments unless they are also specific names.

A generic or specific procedure that has the **INTRINSIC** attribute keeps its generic or specific properties.

A generic intrinsic procedure that has the **INTRINSIC** attribute can also be the name of a generic interface block. The generic interface block defines extensions to the generic intrinsic procedure.

Table 45. Attributes compatible with the INTRINSIC attribute

PRIVATE	PUBLIC

Examples

```
PROGRAM MAIN
  INTRINSIC SIN, ABS
  INTERFACE ABS
    LOGICAL FUNCTION MYABS(ARG)
      LOGICAL ARG
    END FUNCTION
  END INTERFACE

  LOGICAL LANS,LVAR
  REAL(8) DANS,DVAR
  DANS = ABS(DVAR)          ! Calls the DABS intrinsic procedure
  LANS = ABS(LVAR)          ! Calls the MYABS external procedure

! Pass intrinsic procedure name to subroutine
  CALL DOIT(0.5,SIN,X)      ! Passes the SIN specific intrinsic
END PROGRAM

SUBROUTINE DOIT(RIN,OPER,RESULT)
  INTRINSIC :: MATMUL
  INTRINSIC    COS
  RESULT = OPER(RIN)
END SUBROUTINE
```

Related information
- Generic and specific intrinsic procedures are listed in Chapter 14, "Intrinsic procedures," on page 567. See this section to find out if a specific intrinsic name can be used as an actual argument.
- "Generic interface blocks" on page 167

LOGICAL

Purpose

A **LOGICAL** type declaration statement specifies the length and attributes of objects and functions of type logical. Initial values can be assigned to objects.

Syntax

```
►►──LOGICAL────────────────────────────────────entity_decl_list──►◄
              └─kind_selector─┘  ┌─::─────────┐
                               └─,─attr_spec_list─::─┘
```

where:

attr_spec

is any of the following:

ALLOCATABLE [1]	**INTRINSIC**	**PUBLIC**
ASYNCHRONOUS	**OPTIONAL**	**SAVE**
AUTOMATIC [2]	**PARAMETER**	**STATIC** [2]
BIND [1]	**POINTER**	**TARGET**
DIMENSION (*array_spec*)	**PRIVATE**	**VALUE** [1]
EXTERNAL	**PROTECTED** [1]	**VOLATILE**
INTENT (*intent_spec*)		
Note: [1] Fortran 2003 [2] IBM extension		

kind_selector

```
►►──┬──(──────────────int_constant_expr──)──────┬──────────────────────►◄
    │    └─KIND──=─┘                             │
    │                              (1)           │
    └─ * ─int_literal_constant─────────────────────┘
```

Notes:

1   IBM extension.

> **IBM** Specifies the length of logical entities: 1, 2, 4 or 8. *int_literal_constant* cannot specify a kind type parameter. **IBM**

attr_spec

For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec

is either **IN**, **OUT**, or **INOUT**

:: is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
is a list of dimension bounds

entity_decl

```
►►──a─────────────────────────────────────────────────────────────►
         ┌──────────────────┐      ┌─(─array_spec─)─┐
         │      (1)    (2)   │      └────────────────┘
         └─ * ───────len──── ┘
         ┌──────────────────────────┐
         │                    (3)    │
         └─(─array_spec─)─ * ─len────┘

►─────────────────────────────────────────────────────────────────►◄
     ┌────────────────────────────────┐
     │                         (4)     │
     ├─/─initial_value_list─/──────────┤
     ├─ = ─constant_expr───────────────┤
     └─ => ─NULL()─────────────────────┘
```

Notes:

1   IBM extension.

2   IBM extension.

3   IBM extension

4   IBM extension.

---

a
is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

IBM *len*
overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications. **IBM**

IBM *initial_value*
provides an initial value for the entity specified by the immediately preceding name. **IBM**

constant_expr
provides a constant expression for the entity specified by the immediately preceding name.

=> **NULL()**
provides the initial value for the pointer object.

Rules

Within the context of a derived type definition:

- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.

- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a pointer, a function result, an object in blank common, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit or if it appears in a named common block in a module `F2008` or submodule `F2008` .

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined.

If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the ▶F2003 **ALLOCATABLE** or ◀F2003 **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

IBM If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. **IBM**

Examples

```
LOGICAL, ALLOCATABLE :: L(:,:)
LOGICAL :: Z=.TRUE.
```

Related information

- "Logical" on page 43
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

MODULE

Purpose

The **MODULE** statement is the first statement of a module program unit, which contains specifications and definitions that can be made accessible to other program units.

Syntax

```
►►──MODULE──module_name──────────────────────────────────────►◄
```

Rules

The module name is a global entity that is referenced by the **USE** statement in other program units to access the public entities of the module. A user-defined module must not have the same name as any other program unit, external procedure, common block, binding labels of global entities in the program, nor can it be the same as any local name in the module.

If the **END** statement that completes the module specifies a module name, the name must be the same as that specified in the **MODULE** statement.

Examples

```
MODULE MM
   CONTAINS
      REAL FUNCTION SUM(CARG)
         COMPLEX CARG
         SUM_FNC(CARG) = IMAG(CARG) + REAL(CARG)
```

```
          SUM = SUM_FNC(CARG)
          RETURN
        ENTRY AVERAGE(CARG)
          AVERAGE = SUM_FNC(CARG) / 2.0
        END FUNCTION SUM
        SUBROUTINE SHOW_SUM(SARG)
          COMPLEX SARG
          REAL SUM_TMP
   10     FORMAT('SUM:',E10.3,' REAL:',E10.3,' IMAG',E10.3)
          SUM_TMP = SUM(CARG=SARG)
          WRITE(10,10) SUM_TMP, SARG
        END SUBROUTINE SHOW_SUM
END MODULE MM
```

Related information

- "Modules" on page 177
- "USE" on page 483
- "Use association" on page 158
- "END" on page 350, for details on the **END MODULE** statement
- "PRIVATE" on page 432
- F2003 "PROTECTED (Fortran 2003)" on page 438 F2003
- "PUBLIC" on page 440

MODULE PROCEDURE (Fortran 2008)

Purpose

A **MODULE PROCEDURE** statement is the first statement of a separate module subprogram.

Syntax

```
►►──MODULE──PROCEDURE──procedure_name────────────────────────────────►◄
```

Rules

The *procedure_name* must be declared to be a separate module procedure in the containing program unit, the ancestor module, or an ancestor submodule.

Examples

The following example shows how a **MODULE PROCEDURE** statement is used as the first statement of a separate module subprogram.

```
MODULE m
  INTERFACE
    MODULE SUBROUTINE sub(arg)
      INTEGER :: arg
    END SUBROUTINE
  END INTERFACE
END MODULE

SUBMODULE (m) n
  CONTAINS
```

```
      MODULE PROCEDURE sub      ! MODULE PROCEDURE statement
        arg = 1
      END PROCEDURE
END SUBMODULE
```

Related information

- "Separate module procedures (Fortran 2008)" on page 188
- "Separate module subprograms (Fortran 2008)" on page 189
- "END" on page 350

NAMELIST

Purpose

The **NAMELIST** statement specifies one or more lists of names for use in **READ**, **WRITE**, and **PRINT** statements.

Syntax
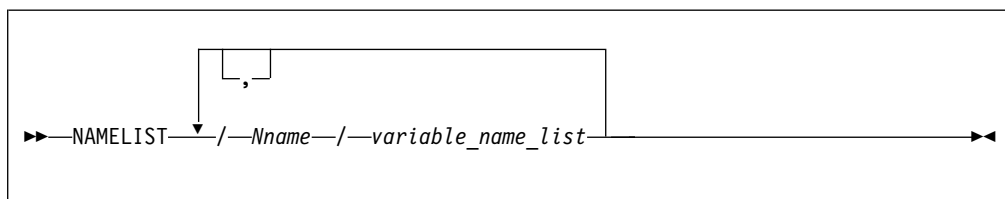
Nname is a namelist group name

variable_name
> Must not be an assumed-size array, or a pointee. If *variable_name* is a variable for a type that has an ultimate component that is a pointer, or an allocatable object, it must be processed by a user-defined derived-type I/O procedure.

Rules

The list of names belonging to a namelist group name ends with the appearance of another namelist group name or the end of the **NAMELIST** statement.

variable_name must either be accessed via use or host association, or have its type and type parameters specified by previous specification statements in the same scoping unit or by the implicit typing rules. If typed implicitly, any appearance of the object in a subsequent type declaration statement must confirm the implied type and type parameters. A derived-type object must not appear as a list item if any component ultimately contained within the object is not accessible within the scoping unit containing the namelist input/output statement on which its containing namelist group name is specified; unless it is processed by a user-defined derived-type input/output procedure.

variable_name can belong to one or more namelist lists. If the namelist group name has the **PUBLIC** attribute, no item in the list can have the **PRIVATE** attribute or private components.

Nname can be specified in more than one **NAMELIST** statement in the scoping unit, and more than once in each **NAMELIST** statement. The *variable_name_list*

following each successive appearance of the same *Nname* in a scoping unit is treated as the continuation of the list for that *Nname*.

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.

Examples

```
DIMENSION X(5), Y(10)
NAMELIST /NAME1/ I,J,K
NAMELIST /NAME2/ A,B,C /NAME3/ X,Y
WRITE (10, NAME1)
PRINT NAME2
```

Related information
- "Namelist formatting" on page 272
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*

NULLIFY

Purpose

The **NULLIFY** statement causes pointers to become disassociated.

Syntax

▶▶──NULLIFY──(──*pointer_object_list*──)──────────────────────────▶◀

pointer_object
     is a pointer variable name or structure component

Rules

A *pointer_object* must be definable and have the **POINTER** attribute.

A *pointer_object* must not depend on the value, bounds, or association status of another *pointer_object* in the same **NULLIFY** statement.

Tip:

Always initialize a pointer with the **NULLIFY** statement, pointer assignment, default initialization or explicit initialization.

Examples

```
TYPE T
  INTEGER CELL
  TYPE(T), POINTER :: NEXT
ENDTYPE T
TYPE(T) HEAD, TAIL
TARGET :: TAIL
HEAD%NEXT => TAIL
NULLIFY (TAIL%NEXT)
END
```

Related information
- "Data pointer assignment" on page 130
- "Pointer association" on page 159

OPEN

Purpose

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.

Syntax

```
▶▶──OPEN──(──open_list──)──────────────────────────────────────────▶◀
```

open_list
> is a list that must contain either one unit specifier ([**UNIT=**_u_]) `F2008` or one **NEWUNIT=** specifier `F2008` . The list can optionally contain one of each of the other valid specifiers. When the list contains more than one specifier, use a comma (,) as the separator. The valid specifiers are as follows:

[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 expression. The integer expression has one of the following values:
> - A value in the range 1 through 2147483647
> - `F2008` A NEWUNIT value `F2008`
>
> If the optional characters **UNIT=** are omitted, *u* must be the first item in *open_list*.

ACCESS= *char_expr*
> specifies the access method for the connection of the file. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **SEQUENTIAL**, **DIRECT** or **STREAM**. If **ACCESS=** is **DIRECT**, **RECL=** must be specified. `F2003` If **ACCESS=** is **STREAM**, **RECL=** must not be specified `F2003` .
>
> **SEQUENTIAL** is the default, for which **RECL=** is optional

ACTION= *char_expr*
> specifies the allowed input/output operations. *char_expr* is a scalar character expression whose value evaluates to **READ**, **WRITE** or **READWRITE**. If **READ** is specified, **WRITE**, **PRINT** and **ENDFILE** statements cannot refer to this connection. If **WRITE** is specified, **READ** statements cannot refer to this connection. The value **READWRITE** permits any input/output statement to refer to this connection. If the **ACTION=** specifier is omitted, the default value depends on the actual file permissions:

- If the **STATUS=** specifier has the value **OLD** or **UNKNOWN** and the file already exists:
  - The file is opened with **READWRITE**
  - If the above is not possible, the file is opened with **READ**
  - If neither of the above is possible, the file is opened with **WRITE**.
- If the **STATUS=** specifier has the value **NEW**, **REPLACE**, **SCRATCH** or **UNKNOWN** and the file does not exist:
  - The file is opened with **READWRITE**
  - If the above is not possible, the file is opened with **WRITE**.

ASYNCH= *char_expr* **(IBM extension)**

is an asynchronous I/O specifier that indicates whether an explicitly connected unit is to be used for asynchronous I/O.

char_expr is a scalar character expression whose value is either **YES** or **NO**. **YES** specifies that asynchronous data transfer statements are permitted for this connection. **NO** specifies that asynchronous data transfer statements are not permitted for this connection. The value specified will be in the set of transfer methods permitted for the file. If this specifier is omitted, the default value is **NO**.

Preconnected units are connected with an **ASYNCH=** value of **NO**.

The **ASYNCH=** value of an implicitly connected unit is determined by the first data transfer statement performed on the unit. If the first statement performs an asynchronous data transfer and the file being implicitly connected permits asynchronous data transfers, the **ASYNCH=** value is **YES**. Otherwise, the **ASYNCH=** value is **NO**.

ASYNCHRONOUS=*char_expr* **(fortran 2003)**

specifies whether or not asynchronous I/O on the unit is allowed.

char_expr is a scalar character expression whose value is either **YES** or **NO**. If *char_expr* is the value **YES** asynchronous I/O on the unit is allowed. If *char_expr* is the value **NO** asynchronous I/O on the unit is not allowed. If **ASYNCHRONOUS=** is not present, the default value is **NO**.

An IBM **ASYNCH=** specifier IBM and an F2003 **ASYNCHRONOUS=** specifier F2003 should not appear on the same **OPEN** statement, the second one is ignored.

BLANK= *char_expr*

controls the default interpretation of blanks when you are using a format specification. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **NULL** or **ZERO**. If **BLANK=** is specified, you must use **FORM='FORMATTED'**. If **BLANK=** is not specified and you specify **FORM='FORMATTED'**, **NULL** is the default.

CONVERT= *char_expr*

specifies the byte order for I/O operations on unformatted data files. *char_expr* is a scalar character expression whose value must evaluate to **NATIVE**, **BIG_ENDIAN**, or **LITTLE_ENDIAN**. If the value is **NATIVE** or **LITTLE_ENDIAN**, little-endian byte order is used and no conversion is necessary. If the value is **BIG_ENDIAN**, the byte order of data and record markers is converted at run time.

You can specify the byte order for I/O operations in the following ways. If more than one is specified for the same unit and they conflict with each other, the one that comes first in the list takes precedence:

1. Runtime option **XLFRTEOPTS**=**ufmt_bigendian**
2. **CONVERT**= *char_expr* in the **OPEN** statement, where *char_expr* is **NATIVE**, **BIG_ENDIAN**, or **LITTLE_ENDIAN**
3. `@PROCESS UFMT(BE)` or `@PROCESS UFMT(LE)`
4. Compiler option **-qufmt=be** or **-qufmt=le**

DECIMAL= *char_expr* **(Fortran 2003)**

specifies the default *decimal edit mode* for the corresponding unit. *char_expr* is a scalar character expression whose value must evaluate to either **POINT** or **COMMA.** If **DECIMAL=** is not specified, the decimal point mode is in effect by default.

DELIM= *char_expr*

specifies what delimiter, if any, is used to delimit character constants written with list-directed or namelist formatting. *char_expr* is a scalar character expression whose value must evaluate to **APOSTROPHE**, **QUOTE**, or **NONE**. If the value is **APOSTROPHE**, apostrophes delimit character constants and all apostrophes within character constants are doubled. If the value is **QUOTE**, double quotation marks delimit character constants and all double quotation marks within character constants are doubled. If the value is **NONE**, character constants are not delimited and no characters are doubled. The default value is **NONE**. The **DELIM=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during input of a formatted record.

ENCODING= *char_expr* **(Fortran 2003)**

specifies the encoding form of the file. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is **DEFAULT**. The **ENCODING=** specifier must only appear in formatted I/O statements. If omitted, the default value is **DEFAULT**.

ERR= *stmt_label*

is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

FILE= *char_expr*

is a file specifier that specifies the name of the file to be connected to the specified unit. `IBM`

char_expr is a scalar character expression whose value, when any trailing blanks are removed, is a valid Linux operating system file name. If the file specifier is omitted and is required, the unit becomes implicitly connected (by default) to **fort.***u*, where *u* is the unit specified with any leading zeros removed. Use the **UNIT_VARS** run-time option to allow alternative files names to be used for files that are implicitly connected.

Note: A valid Linux operating system file name must have a full path name of total length ≤1023 characters, with each file name ≤255 characters long (although the full path name need not be specified).
`IBM`

FORM= *char_expr*

specifies whether the file is connected for formatted or unformatted input/output. *char_expr* is a scalar character expression whose value, when

any trailing blanks are removed, is either **FORMATTED** or **UNFORMATTED**. If you connect the file for sequential access, **FORMATTED** is the default. If you connect the file for direct access ▸F2003▸ or stream access ◂F2003◂, **UNFORMATTED** is the default.

IOMSG= *iomsg_variable* **(Fortran 2003)**

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

An input/output status specifier for the status of the input/output operation. *ios* is a scalar integer variable. When the input/output statement containing this specifier finishes execution, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs

NEWUNIT= *var* **(Fortran 2008)**

an input/output specifier that specifies the NEWUNIT value for the connection. *var* is a scalar default integer variable. The NEWUNIT value is a negative number that is less than -2 and is unequal to the unit number of any currently connected file. If you specify **NEWUNIT=** specifier in the **OPEN** statement, you must also specify the **STATUS=** specifier with value **SCRATCH** or specify the **FILE=** specifier.

PAD= *char_expr*

specifies if input records are padded with blanks. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. If the value is **YES**, a formatted input record is padded with blanks if an input list is specified and the format specification requires more data from a record than the record contains. If **NO** is specified, the input list and format specification must not require more characters from a record than the record contains. The default value is **YES**. The **PAD=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during output of a formatted record.

IBM If the **-qxlf77** compiler option specifies the **noblankpad** suboption and the file is being connected for formatted direct input/output, the default value is **NO** when the **PAD=** specifier is omitted.**IBM**

POSITION= *char_expr*

specifies the file position for a file connected for sequential or stream access. A file that did not exist previously is positioned at its initial point. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **ASIS**, **REWIND**, or **APPEND**. **REWIND** positions the file at its initial point. **APPEND** positions the file before the endfile record or, if there is no endfile record, at the terminal point. **ASIS** leaves the position unchanged. The default value is **ASIS** except under the following conditions:

- The first input/output statement (other than the **INQUIRE** statement) referring to the unit after the **OPEN** statement is a **WRITE** statement, and either:

- The **STATUS=** specifier is **UNKNOWN** and the **-qposition** compiler option specifies **appendunknown**, or
- The **STATUS=** specifier is **OLD** and the **-qposition** compiler option specifies **appendold**.

In such cases, the default value for the **POSITION=** specifier is **APPEND** at the time the **WRITE** statement is executed.

RECL= *integer_expr*

specifies the length of each record in a file being connected for direct access or the maximum length of a record in a file being connected for sequential access. *integer_expr* is an integer expression whose value must be positive. This specifier must be present when a file is being connected for direct access. For formatted input/output, the length is the number of characters for all records that contain character data. For unformatted input/output, the length is the number of bytes required for the internal form of the data. The length of an unformatted sequential record does not count the four-byte fields surrounding the data.

IBM

For units connected to files that cannot be accessed randomly, the default length is 2**15 (32,768).

If **RECL=** is omitted when a file is being connected for sequential access, the length is 2**63-1, minus the record terminator. For a formatted sequential file, the default record length is 2**63-2. For an unformatted file, the default record length is 2**63-17 when the **UWIDTH** run-time option is set to 64.

IBM

ROUND= *char_expr* **(Fortran 2003)**

states the current value of the I/O rounding mode for formatted input and output. The **ROUND=** can be changed by other statements. If omitted, then the processor can choose the rounding mode. *char_expr* evaluates to either **UP**, **DOWN**, **ZERO**, **NEAREST**, **COMPATIBLE** or **PROCESSOR_DEFINED**

The rounding mode helps specify how decimal numbers are converted to an internal representation, (that is, in binary) from a character representation and vice versa during formatted input and output. The rounding modes have the following functions:

- In the **UP** rounding mode the value from the conversion is the smallest value that is greater than or equal to the original value.
- In the **DOWN** rounding mode the value from the conversion is the greatest value that is smaller than or equal to the original value.
- In the **ZERO** rounding mode the value from the conversion is the closest value to the original value, and not greater in magnitude.
- In the **NEAREST** rounding mode the value from the conversion is the closer of the two nearest representable values. If both values are equally close then the even value will be chosen. In IEEE rounding conversions, NEAREST corresponds to the ieee_nearest rounding mode as specified by the IEEE standard.

- In the **COMPATIBLE** rounding mode the value from the conversion is the closest of the two nearest representable values, or the value further away from zero if halfway between.
- In the **PROCESSOR_DEFINED** rounding mode the value from the conversion is processor-dependent and may correspond to the other modes. In XL Fortran, the **PROCESSOR_DEFINED** rounding mode will be the rounding mode you choose in the floating-point control register. If you do not set the floating-point control register explicitly, the default rounding mode is **NEAREST**.

SIGN= *char_expr* **(Fortran 2003)**

indicates the sign mode in effect for a connection for formatted input/output. If *char_expr* is assigned the value **PLUS**, the processor shall produce a plus sign in any position that normally contains an optional plus sign and suppresses plus signs in these positions if *char_expr* is assigned the value **SUPPRESS**. *char_expr* can also be assigned the value **PROCESSOR_DEFINED** which is the default sign mode and acts the same as **SUPPRESS**. If there is no connection, or if the connection is not for formatted input/output, *char_expr* is assigned the value **UNDEFINED**. The sign mode may be temporarily changed in a single data transfer statement. When the statement terminates, the sign mode resumes its previous value.

STATUS= *char_expr*

specifies the status of the file when it is opened. *char_expr* is a scalar 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, the file must exist. If the file does not exist, an error condition will occur.
- **NEW**, to create a new file, connect it to a unit, and change the status to **OLD**. If **NEW** is specified, the file must not exist. If the file already exists, an error condition will occur.
- **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).
- **REPLACE**. If the file does not already exist, the file is created and the status is changed to **OLD**. If the file exists, the file is deleted, a new file is created with the same name, and the status is changed to **OLD**.
- **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.

Rules

If a unit is connected to a file that exists, an **OPEN** statement for that unit can 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.

F2008 If an **OPEN** statement containing a **NEWUNIT=** specifier is executed successfully, the variable specified by **NEWUNIT=** is assigned with a new NEWUNIT value. However, if an error occurs during the execution of the **OPEN** statement, the variable specified by **NEWUNIT=** keeps its original value. F2008

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=** 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=**, **DELIM=**, **PAD=**, **ERR=**, and **IOSTAT=** specifiers can have a value different from the one currently in effect. Execution of the **OPEN** statement causes any new value for the **BLANK=**, **DELIM=** or **PAD=** specifiers to be in effect, but does not cause any change in any of the unspecified specifiers or the position of the file. Any **ERR=** and **IOSTAT=** specifiers from **OPEN** statements previously executed have no effect on the current **OPEN** statement. If you specify the **STATUS=** specifier it must have the value **OLD**. To specify the same file as the one currently connected to the unit, you can specify the same file name, omit the **FILE=** specifier, or specify a file symbolically linked to the same file.

If a file is connected to a unit, an **OPEN** statement on that file and a different unit cannot be performed.

IBM If the **STATUS=** specifier has the value **OLD**, **NEW** or **REPLACE**, the **FILE=** specifier is optional.

Unit 0 cannot be specified to connect to a file other than the preconnected file, the standard error device, although you can change the values for the **BLANK=**, **DELIM=** and **PAD=** specifiers. IBM

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

IBM If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops. IBM

Examples

F2008

Example 1: write 'hello world' to file 'hello'
```
INTEGER unit_number
OPEN(newunit = unit_number, file = 'hello')
WRITE(unit_number, *) 'hello world'
CLOSE(unit_number)
```

F2008

Example 2:
```
!   Open a new file with name fname

CHARACTER*20 FNAME
FNAME = 'INPUT.DAT'
OPEN(UNIT=8,FILE=FNAME,STATUS='NEW',FORM='FORMATTED')

OPEN (4,FILE="myfile")
OPEN (4,FILE="myfile", PAD="NO")  ! Changing PAD= value to NO
```

```
!  Connects unit 2 to a tape device for unformatted, sequential
!  write-only access:

OPEN (2, FILE="/dev/rmt0",ACTION="WRITE",POSITION="REWIND", &
&   FORM="UNFORMATTED",ACCESS="SEQUENTIAL",RECL=32767)
```

Related information

- "Units" on page 218
- Item 3 under "Compatibility across standards" on page 1051
- Chapter 9, "XL Fortran Input/Output," on page 215
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*
- The XLFRTEOPTS environment variable in the *XL Fortran Compiler Reference*
- **-qposition** option in the *XL Fortran Compiler Reference*
- **-qxlf77** option in the *XL Fortran Compiler Reference*
- "CLOSE" on page 313
- "READ" on page 441
- "WRITE" on page 495

OPTIONAL

Purpose

The **OPTIONAL** attribute specifies that a dummy argument need not be associated with an actual argument in a reference to the procedure.

Syntax

```
►►──OPTIONAL──┬────┬──dummy_arg_name_list─────────────────────►◄
              └─::─┘
```
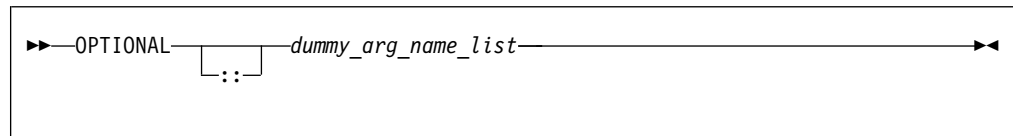
Rules

A procedure that has an optional dummy argument must have an explicit interface in any scope in which the procedure is referenced.

Use the **PRESENT** intrinsic function to determine if an actual argument has been associated with an optional dummy argument. Avoid referencing an optional dummy argument without first verifying that the dummy argument is present.

A dummy argument is considered present in a subprogram according to the rules described in the section: "Restrictions on optional dummy arguments not present" on page 198.

An optional dummy argument that is not present may be used as an actual argument corresponding to an optional dummy argument, which is then also considered not to be associated with an actual argument. An optional dummy argument that is not present is subject to the restrictions specified in the section: "Restrictions on optional dummy arguments not present" on page 198

The **OPTIONAL** attribute cannot be specified for dummy arguments in an interface body that specifies an explicit interface for a defined operator or defined assignment.

Table 46. Attributes compatible with the OPTIONAL attribute

ALLOCATABLE **1**	EXTERNAL	TARGET
ASYNCHRONOUS	INTENT	VALUE **1**
CONTIGUOUS **2**	POINTER	VOLATILE
DIMENSION		
Note: **1** Fortran 2003 **2** Fortran 2008		

Notes:

1. Fortran 2008.

Examples

```
SUBROUTINE SUB (X,Y)
  INTERFACE
    SUBROUTINE SUB2 (A,B)
      OPTIONAL :: B
    END SUBROUTINE
  END INTERFACE
  OPTIONAL :: Y
  IF (PRESENT(Y)) THEN          ! Reference to Y conditional
    X = X + Y                    ! on its presence
  ENDIF
  CALL SUB2(X,Y)
END SUBROUTINE

SUBROUTINE SUB2 (A,B)
  OPTIONAL :: B                 ! B and Y are argument associated,
  IF (PRESENT(B)) THEN          ! even if Y is not present, in
    B = B * A                   ! which case, B is also not present
    PRINT*, B
  ELSE
    A = A**2
    PRINT*, A
  ENDIF
END SUBROUTINE
```

Related information

- "Optional dummy arguments" on page 198
- "Interface concepts" on page 162
- "PRESENT(A)" on page 679
- "Dummy arguments" on page 194

PARAMETER

Purpose

The **PARAMETER** attribute allows you to specify names for constants.

Syntax

```
                    ,
   ┌──────────────────────────┐
▶▶─PARAMETER─(──▼─constant_name─ = ─constant_expr──┘──)────────────▶◀
```
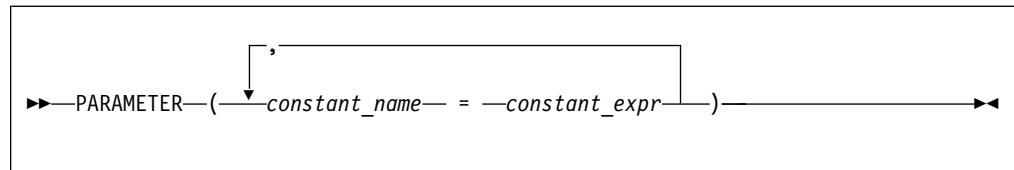
constant_expr
> A constant expression

Rules

A named constant must have its type, shape, and parameters specified in a previous specification statement in the same scoping unit or be declared implicitly. If a named constant is implicitly typed, its appearance in any subsequent type declaration statement or attribute specification statement must confirm the implied type and any parameter values.

You can define *constant_name* only once with a **PARAMETER** attribute in a scoping unit.

A named constant that is specified in the constant expression must have been previously defined (possibly in the same **PARAMETER** or type declaration statement, if not in a previous statement) or made accessible through use or host association.

The constant expression is assigned to the named constant using the rules for intrinsic assignment. If the named constant is of type character and it has inherited length, it takes on the length of the constant expression.

Table 47. Attributes compatible with the PARAMETER attribute

DIMENSION	PRIVATE	PUBLIC

Examples

```
REAL, PARAMETER :: TWO=2.0

COMPLEX     XCONST
REAL        RPART,IPART
PARAMETER   (RPART=1.1,IPART=2.2)
PARAMETER   (XCONST = (RPART,IPART+3.3))

CHARACTER*2, PARAMETER :: BB='   '
   ...
END
```

Related information
- "Constant expressions" on page 102
- "Data objects" on page 19

PAUSE

Purpose

The **PAUSE** statement temporarily suspends the execution of a program and prints the keyword **PAUSE** and, if specified, a character constant or digit string to unit 0.

Syntax

```
►►──PAUSE──┬────────────────────┬──────────────────────────────►◄
           ├──char_constant─────┤
           └──digit_string──────┘
```
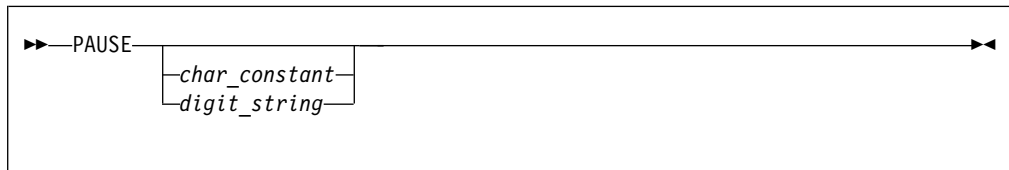
char_constant
> is a scalar character constant that is not a Hollerith constant

digit_string
> is a string of one to five digits

Rules

IBM After execution of a **PAUSE** statement, processing continues when you press the **Enter** key. If unit 5 is not connected to the terminal, the **PAUSE** statement does not suspend execution. **IBM**

The **PAUSE** statement has been deleted in Fortran 95.

Examples

```
PAUSE 'Ensure backup tape is in tape drive'
PAUSE 10               ! Output:  PAUSE 10
```

Related information

POINTER (Fortran 90)

Purpose

The **POINTER** attribute designates objects as pointer variables.

The term *pointer* refers to objects with the Fortran 90 **POINTER** attribute. The integer **POINTER** statement provides details on what was documented in previous versions of XL Fortran as the **POINTER** statement; these pointers are now referred to as *integer pointers*.

Syntax

```
                       ┌───────,───────────────────┐
                       │                            │
►►──POINTER──┬──────┬──▼──object_name──┬──────────────────────────────┬──┴──►◄
             └──::──┘                  └──(──deferred_shape_spec_list──)──┘
```
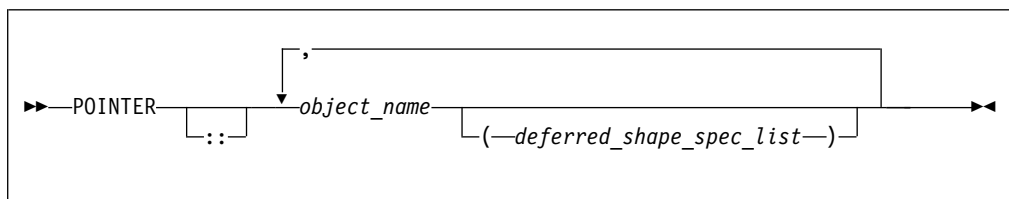
deferred_shape_spec
> is a colon (:), where each colon represents a dimension

Rules

object_name refers to a data object or function result. If *object_name* is declared elsewhere in the scoping unit with the **DIMENSION** attribute, the array specification must be a *deferred_shape_spec_list*.

object_name must not appear in an integer **POINTER**, **NAMELIST**, or **EQUIVALENCE** statement. If *object_name* is a component of a derived-type definition, any variables declared with that type cannot be specified in an **EQUIVALENCE** or **NAMELIST** statement.

Pointer variables can appear in common blocks and block data program units.

IBM To ensure that Fortran 90 pointers are thread-specific, do not specify either the **SAVE** or **STATIC** attribute for the pointer. These attributes are either specified explicitly by the user, or implicitly through the use of the **–qsave** compiler option. Note, however, that if a non-static pointer is used in a pointer assignment statement where the target is static, all references to the pointer are, in fact, references to the static, shared target. **IBM**

An object having a component with the **POINTER** attribute can itself have the **TARGET**, **INTENT**, or **ALLOCATABLE** attibutes, although it cannot appear in a data transfer statement.

TS 29113 You can specify the **POINTER** attribute for assumed-rank entities. **TS 29113**

Table 48. Attributes compatible with the POINTER attribute

AUTOMATIC **3**	INTENT	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE
CONTIGUOUS **2**	PRIVATE	STATIC **3**
DIMENSION	PROTECTED **1**	VOLATILE
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

These attributes apply only to the pointer itself, not to any associated targets, except for the **DIMENSION** attribute, which applies to associated targets.

Examples

Example1:
```
INTEGER, POINTER :: PTR(:)
INTEGER, TARGET :: TARG(5)
PTR => TARG                    ! PTR is associated with TARG and is
                               !   assigned an array specification of (5)

PTR(1) = 5                     ! TARG(1) has value of 5
PRINT *, FUNC()
CONTAINS
  REAL FUNCTION FUNC()
    POINTER :: FUNC            ! Function result is a pointer

       .
```

```
        .
        .
  END FUNCTION
END
```

Example 2: Fortran 90 pointers and threadsafing

```
FUNCTION MYFUNC(ARG)                ! MYPTR is thread-specific.
INTEGER, POINTER :: MYPTR           ! every thread that invokes
                                    !  'MYFUNC' will allocate a
ALLOCATE(MYPTR)                     !  new piece of storage that
MYPTR = ARG                         !  is only accessible within
                                    !  that thread.
ANYVAR = MYPTR
END FUNCTION
```

Related information

- "Data pointer assignment" on page 130
- "TARGET" on page 471
- "ALLOCATED(X)" on page 580
- "DEALLOCATE" on page 331
- "Pointer association" on page 159
- "Deferred-shape arrays" on page 80
- "Interoperability of procedures" on page 888

POINTER (integer) (IBM extension)

Purpose

The integer **POINTER** statement specifies that the value of the variable *int_pointer* is to be used as the address for any reference to *pointee*.

The name of this statement has been changed from **POINTER** to integer **POINTER** to distinguish it from the Fortran 90 **POINTER** statement.

Syntax

```
                  ┌─────,──────┐
                  │            │
▶▶──POINTER──▼─(──int_pointer──,──pointee──)──┴────────────────────▶◀
```
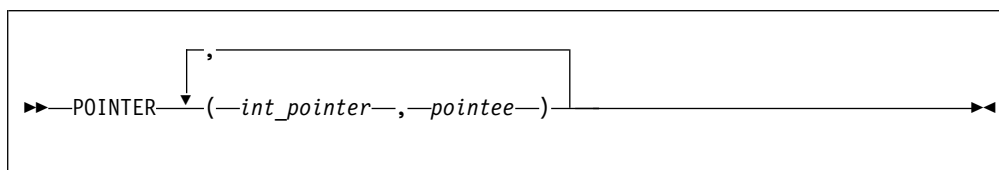
int_pointer
    is the name of an integer pointer variable

pointee   is a variable name or array declarator

Rules

The compiler does not allocate storage for the pointee. Storage is associated with the pointee at execution time by the assignment of the address of a block of

storage to the pointer. The pointee can become associated with either static or dynamic storage. A reference to a pointee requires that the associated pointer be defined.

An integer pointer is a scalar variable of type **INTEGER(8)** that cannot have a type explicitly assigned to it. You can use integer pointers in any expression or statement in which a variable of the same type as the integer pointer can be used. You can assign any data type to a pointee, but you cannot assign a storage class or initial value to a pointee.

An actual array that appears as a pointee in an integer **POINTER** statement is called a pointee array. You can dimension a pointee array in a type declaration statement, a **DIMENSION** statement, or in the integer **POINTER** statement itself.

If you specify the **-qddim** compiler option, a pointee array that appears in a main program can also have an adjustable array specification. In main programs and subprograms, the dimension size is evaluated when the pointee is referenced (dynamic dimensioning).

If you do not specify the **-qddim** compiler option, a pointee array that appears in a subprogram can have an adjustable array specification, and the dimension size is evaluated on entrance to the subprogram, not when the pointee is evaluated.

The following constraints apply to the definition and use of pointees and integer pointers:

- A pointee cannot be zero-sized.
- A pointee can be scalar, an assumed-sized array or an explicit-shape array.
- A pointee cannot appear in a **COMMON**, **DATA**, **NAMELIST**, or **EQUIVALENCE** statement.
- A pointee cannot have the following attributes: **EXTERNAL**, **ALLOCATABLE**, **POINTER**, **TARGET**, **INTRINSIC**, **INTENT**, **OPTIONAL**, **SAVE**, **STATIC**, **AUTOMATIC**, or **PARAMETER**.
- A pointee cannot be a dummy argument and therefore cannot appear in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.
- A pointee cannot be an automatic object, though a pointee can have nonconstant bounds or lengths.
- A pointee cannot be a generic interface block name.
- A pointee that is of derived type must be of sequence derived type.
- A function value cannot be a pointee.
- An integer pointer cannot be pointed to by another pointer. (A pointer cannot be a pointee.)
- An integer pointer cannot have the following attributes:
  - F2003 **ALLOCATABLE** F2003
  - **DIMENSION**
  - **EXTERNAL**
  - **INTRINSIC**
  - **PARAMETER**
  - **POINTER**
  - **TARGET**
- An integer pointer cannot appear as a **NAMELIST** group name.
- An integer pointer cannot be a procedure.

Examples

```
INTEGER A,B
POINTER (P,I)
IF (A<>0) THEN
  P=LOC(A)
ELSE
  P=LOC(B)
ENDIF
I=0        ! Assigns 0 to either A or B, depending on A's value
END
```

Related information

- "Integer pointer association (IBM extension)" on page 161
- "LOC(X) (IBM extension)" on page 648
- **-qddim option** in the *XL Fortran Compiler Reference*

PRINT

Purpose

The **PRINT** statement is a data transfer output statement.

Syntax

```
►►──PRINT──┬─name────────────────────────────────┬──►◄
           └─format──┬──────────────────────────┬─┘
                     └─,──output_item_list───────┘
```
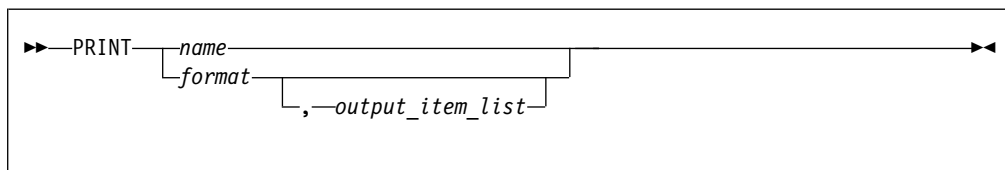
name   is a namelist group name

output_item

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

- A variable. An array is treated as if all of its elements were specified in the order they are arranged in storage.

  A pointer must be associated with a target, and an allocatable object must be allocated. A derived-type object cannot have any ultimate component that is inaccessible to this statement. The evaluation of *output_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a formatted statement are treated as if they appear in the order of the derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).

- An expression.
- An implied-**DO** list, as described under "Implied-DO List" on page 431.

  F2003 An expression that is an *output_item* cannot have a value that is a procedure pointer. F2003

format   is a format specifier that specifies the format to be used in the output operation. *format* is a format identifier that can be:
- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.
- A character constant. It cannot be a Hollerith constant. 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.
- A character variable 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 listed under "FORMAT" on page 376 can be used between the parentheses. Blank characters can precede the left parenthesis, or follow the right parenthesis.
- An array of noncharacter intrinsic type.
- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.
- A namelist specifier that specifies a previously defined namelist.

Specifying the **–qport=typestmt** compiler option enables the **TYPE** statement which has identical functionality to the **PRINT** statement.

Examples

```
   PRINT 10, A,B,C
10 FORMAT (E4.2,G3.2E1,B3)
```

Related information

- Chapter 9, "XL Fortran Input/Output," on page 215
- Chapter 10, "Input/Output formatting," on page 237
- See the *XL Fortran Compiler Reference* for more information on **-qport=typestmt**.
- "Deleted features" on page 1054
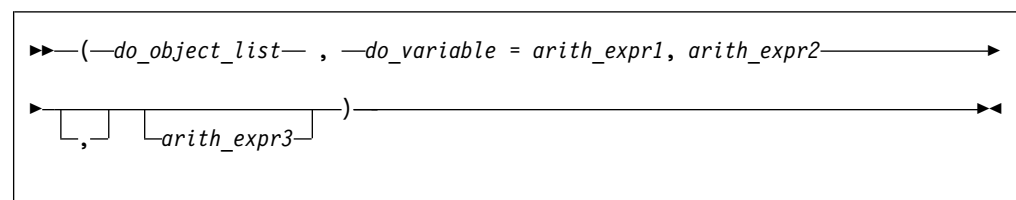
Implied-DO List

```
►►──(──do_object_list── , ──do_variable = arith_expr1, arith_expr2────────►

►──────────────────────)──────────────────────────────────────────────►◄
   └─,─┘  └─arith_expr3─┘
```

do_object
> is an output list item

do_variable
> is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, **and** *arith_expr3*
> are scalar numeric expressions

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. 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.

PRIVATE

Purpose

The **PRIVATE** attribute specifies that a module entity is not accessible outside the module through use association.

Syntax

```
►►──PRIVATE──────────────────────────────────────────────►◄
              │      ┌──────────────┐   │
              │      │              │   │
              └──────┬────────access_id_list─┘
                     └──::──┘
```

access_id
> is a generic specification or the name of a variable, procedure, derived type, constant, or namelist group

Rules

The **PRIVATE** attribute can appear only in the scope of a module.

Although multiple **PRIVATE** statements can appear in a module, you can only include one statement that omits an *access_id_list*. A **PRIVATE** statement without an *access_id_list* sets the default accessibility to private for all potentially accessible entities in the module. If the module contains such a statement, it must not include a **PUBLIC** statement without an *access_id_list*. If the module does not contain a **PRIVATE** statement without an *access_id_list*, the default accessibility is public. Entities whose accessibility is not explicitly specified have default accessibility.

A procedure that has a generic identifier that is public is accessible through that identifier, even if its specific identifier is private. If a module procedure contains a private dummy argument or function result whose type has private accessibility, the module procedure must be declared to have private accessibility and must not have a generic identifier that has public accessibility. The accessibility of a derived type does not affect, and is not affected by, the accessibility of its components or procedures.

A namelist group must be private if it contains any object that is private or contains private components. A subprogram must be private if any of its arguments are of a derived type that is private. A function must be private if its result variable is of a derived type that is private.

Table 49. Attributes compatible with the PRIVATE attribute

ALLOCATABLE **1**	INTRINSIC	SAVE
ASYNCHRONOUS	PARAMETER	STATIC **3**
CONTIGUOUS **2**	POINTER	TARGET

Table 49. Attributes compatible with the PRIVATE attribute (continued)

DIMENSION	PROTECTED **1**	VOLATILE
EXTERNAL		
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

```
MODULE MC
   PUBLIC                    ! Default accessibility declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1, SUB2
   END INTERFACE
   PRIVATE SUB1              ! SUB1 declared as private
   CONTAINS
      SUBROUTINE SUB1(I)
         INTEGER I
         I = I + 1
      END SUBROUTINE SUB1
      SUBROUTINE SUB2(I,J)
         I = I + J
      END SUBROUTINE
END MODULE MC

PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)               ! SUB1 referenced because GEN has public
                             ! accessibility and appropriate argument
                             ! is passed
   CALL SUB2(K,4)
   PRINT *, K                ! Value printed is 10
END PROGRAM
```

Related information

- Chapter 4, "Derived types," on page 49
- "Modules" on page 177
-
- "PUBLIC" on page 440

PROCEDURE

Purpose

A **PROCEDURE** statement that appears within a generic interface adds the specified procedures to the generic interface.

Syntax

```
►►─┬────────┬─PROCEDURE─┬───────────┬─procedure_name_list────────────►◄
   └─MODULE─┘           │    (1)    │
                        └─::────────┘
```

Notes:

1   Fortran 2008
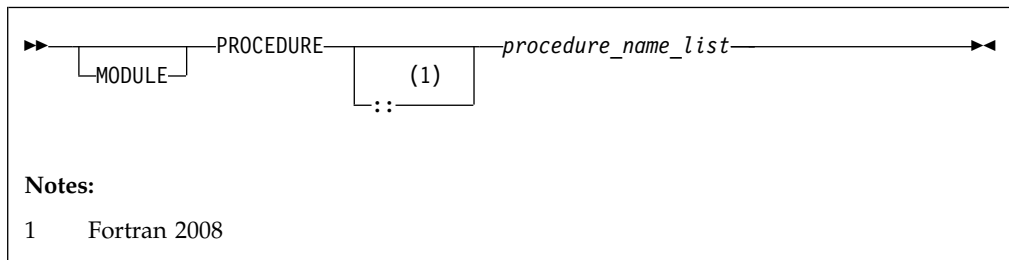
Rules

A **MODULE PROCEDURE** statement can appear anywhere among the interface bodies in an interface block that has a generic specification.

F2003

A **PROCEDURE** statement can only appear in an interface block that has a generic specification.

When **MODULE** is not specified, *procedure_name_list* must refer to an accessible procedure pointer, external procedure, dummy procedure, or module procedure that has an explicit interface. When **MODULE** is specified for **PROCEDURE**, *procedure_name_list* must refer to a module procedure that is accessible in the current scope.

A *procedure_name* must not specify a procedure that is specified previously in any **PROCEDURE** statement in any accessible interface with the same generic identifier.

F2003

Examples

The following example shows how to declare and use a **PROCEDURE** statement in a generic interface:

```
MODULE m
  CONTAINS
  SUBROUTINE s1(iarg)
    iarg=1
    PRINT *, "In s1"
  END SUBROUTINE
  SUBROUTINE s2(rarg)
    rarg=1.1
  END SUBROUTINE
END MODULE

USE m
INTERFACE ss
  SUBROUTINE ss1(iarg,jarg)
  END SUBROUTINE
  MODULE PROCEDURE s1, s2
END INTERFACE
CALL ss(n)                  ! Calls subroutine s1 from m
CALL ss(i,j)                ! Calls subroutine ss1
```

```
END
SUBROUTINE SS1(iarg,jarg)
   PRINT *, "In ss1"
END SUBROUTINE ss1
```

Related information

- "Interface blocks" on page 164
- "INTERFACE" on page 405
- "Modules" on page 177

PROCEDURE declaration (Fortran 2003)

Purpose

A **PROCEDURE** declaration statement declares a dummy procedure, an external
procedure, or a procedure pointer. It specifies the **EXTERNAL** attribute for these
entities.

Syntax
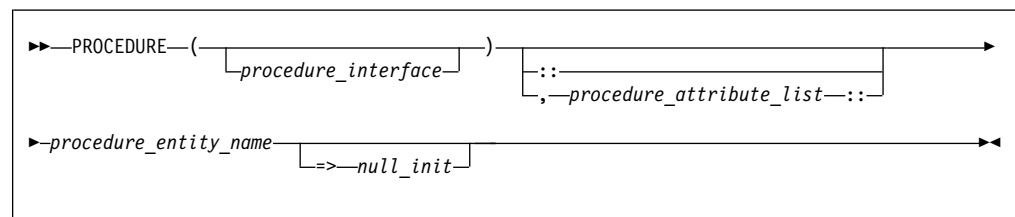
procedure_interface
>   A declaration type specification or the name of a procedure that has an
>   explicit interface.

procedure_attribute_list
>   A list of attributes from the following list:

- **BIND**
- **INTENT**(*intent_spec*)
- **OPTIONAL**
- **POINTER**
- **PRIVATE**
- **PUBLIC**
- **SAVE**

procedure_entity_name
>   is the name of the procedure or procedure pointer that is being declared.

null_init
>   is a reference to the NULL intrinsic function.

Rules

If *procedure_interface* is the name of a procedure or procedure pointer that has an
explicit interface, the declared procedures or procedure pointers have this explicit
interface. The *procedure_interface* must already be declared. The name of the
procedure_interface cannot be the same as a keyword that specifies an intrinsic type.

The *procedure_interface* can be an intrinsic procedure as long as the intrinsic procedure can be passed as an actual argument. If the *procedure_interface* is an elemental procedure, the procedure entity names must consist of external procedures.

If *procedure_interface* is a declaration type specification, the declared procedures or procedure pointers are functions with an implicit interface and the specified result type. If these functions are external functions, the function definitions must specify the same result type and type parameters.

If no *procedure_interface* is specified, the **PROCEDURE** declaration statement specifies that the declared procedures or procedure pointers are either subroutines or functions. If they are functions, the implicit type rule applies to the type of the function.

If you specify procedure language binding using the **BIND** attribute, *procedure_interface* must be the name of a procedure or procedure pointer that is declared with procedure language binding.

If procedure language binding with **NAME=** is specified, the procedure entity name must consist of only one procedure entity name. This procedure must not be a dummy procedure or have the **POINTER** attribute.

If **OPTIONAL** is specified, the declared procedures or procedure pointers must be dummy procedures or procedure pointers.

You can only specify **PUBLIC** or **PRIVATE** if the statement appears in the specification part of a module.

If **INTENT**, **SAVE**, or *null_init* is specified, the declared entities must have the **POINTER** attribute.

If *null_init* is used, it specifies that the initial association status of the corresponding procedure pointer is disassociated. It also implies the **SAVE** attribute, which can be reaffirmed by explicitly using the **SAVE** attribute in the procedure declaration statement or by a **SAVE** statement.

For procedure pointer declarations, you must specify the **POINTER** attribute.

Examples

Example 1

The following example shows an external procedure declaration.
```
CONTAINS
SUBROUTINE XXX(PSI)
    PROCEDURE (REAL) :: PSI
    REAL  Y1
    Y1 = PSI()
END SUBROUTINE
END
```

Example 2

The following example shows a procedure pointer declaration and its use.

```
PROGRAM PROC_PTR_EXAMPLE
  REAL :: R1
  INTEGER :: I1
  INTERFACE
    SUBROUTINE SUB(X)
      REAL, INTENT(IN) :: X
    END SUBROUTINE SUB
    FUNCTION REAL_FUNC(Y)
      REAL, INTENT(IN) :: Y
      REAL, REAL_FUNC
    END FUNCTION REAL_FUNC
  END INTERFACE
  ! with explicit interface
  PROCEDURE(SUB), POINTER :: PTR_TO_SUB
  ! with explicit interface
  PROCEDURE(REAL_FUNC), POINTER :: PTR_TO_REAL_FUNC => NULL()
  ! with implicit interface
  PROCEDURE(INTEGER), POINTER :: PTR_TO_INT
  PTR_TO_SUB => SUB
  PTR_TO_REAL_FUNC => REAL_FUNC
  CALL PTR_TO_SUB(1.0)
  R1 = PTR_TO_REAL_FUNC(2.0)
  I1 = PTR_TO_INT(M, N)
END PROGRAM PROC_PTR_EXAMPLE
```

Related information

- "BIND (Fortran 2003)" on page 298
- Chapter 14, "Intrinsic procedures," on page 567
- Chapter 15, "Hardware-specific intrinsic procedures (IBM extension)," on page 725
- "Program units, procedures, and subprograms" on page 161
- "Intrinsic procedures" on page 191
- "INTERFACE" on page 405
- "Procedure pointer assignment (Fortran 2003)" on page 133

PROGRAM

Purpose

The **PROGRAM** statement specifies that a program unit is a main program, the program unit that receives control from the system when the executable program is invoked at run time.

Syntax

```
►►──PROGRAM──name──────────────────────────────────────────────────►◄
```

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

Rules

The **PROGRAM** statement is optional.

If specified, the **PROGRAM** statement must be the first statement of the main program.

If a program name is specified in the corresponding **END** statement, it must match *name*.

The program name is global to the executable program. This name must not be the same as the name of any common block, external procedure, or any other program unit in that executable program, or as any name that is local to the main program.

The name has no type, and it must not appear in any type declaration or specification statements. You cannot refer to a main program from a subprogram or from itself.

Examples

```
PROGRAM DISPLAY_NUMBER_2
   INTEGER A
   A = 2
   PRINT *, A
END PROGRAM DISPLAY_NUMBER_2
```

Related information

- "END" on page 350
- "Main program" on page 176

PROTECTED (Fortran 2003)

Purpose

The **PROTECTED** attribute allows greater control over the modification of module entities. A module procedure can only modify a protected module entity or its subobjects if the same module defines both the procedure and the entity.

Syntax

The **PROTECTED** attribute must only appear in the specification part of the module.

```
►►──PROTECTED──────────entity_declaration_list────────────────────►◄
                └─::─┘
```
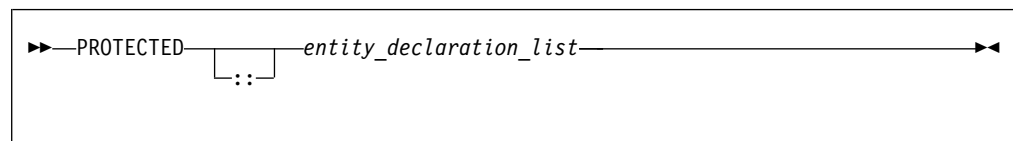
entity   A named variable not in a common block.

Rules

If you specify that an object declared by an **EQUIVALENCE** statement has the **PROTECTED** attribute, all objects specified in that **EQUIVALENCE** statement must have the **PROTECTED** attribute.

A nonpointer object with the **PROTECTED** attribute accessed through use association, is not definable.

You must not specify the **PROTECTED** attribute for integer pointers.

A pointer object with the **PROTECTED** attribute accessed through use association, must not appear as any of the following:

- As a pointer object in a **NULLIFY** statement or **POINTER** assignment statement
- As an allocatable object in an **ALLOCATE** or **DEALLOCATE** statement.
- As an actual argument in reference to a procedure, if the associated dummy argument is a pointer with the **INTENT(INOUT)** or **INTENT(OUT)** attribute.

Table 50. Attributes compatible with the PROTECTED attribute

ALLOCATABLE **1**	INTENT	SAVE
ASYNCHRONOUS	OPTIONAL	STATIC **3**
AUTOMATIC **3**	POINTER	TARGET
CONTIGUOUS **2**	PRIVATE	VOLATILE
DIMENSION	PUBLIC	
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

In the following example, the values of both *age* and *val* can only be modified by subroutines in the module in which they are declared:

```
module mod1
    integer, protected :: val
    integer :: age
    protected :: age
    contains
        subroutine set_val(arg)
            integer arg
             val = arg
        end subroutine
         subroutine set_age(arg)
            integer arg
             age  = arg
        end subroutine
end module
program dt_init01
    use mod1
    implicit none
    integer :: value, his_age
    call set_val(88)
    call set_age(38)
    value = val
    his_age = age
    print *, value, his_age
end program
```

Related information

- "Modules" on page 177
- "PRIVATE" on page 432
- "PUBLIC" on page 440

PUBLIC

Purpose

The **PUBLIC** attribute specifies that a module entity can be accessed by other program units through use association.
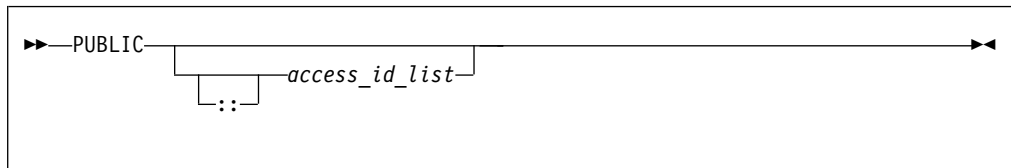
Syntax

```
>>--PUBLIC---------------------------------------------------><
            |            ,-----------------|
            |  |--access_id_list-|         |
            '--:-|             |-----------'
                '-::-'
```

access_id

is a generic specification or the name of a variable, procedure, derived type, constant, or namelist group

Rules

The **PUBLIC** attribute can appear only in the scope of a module.

Although multiple **PUBLIC** statements can appear in a module, only one statement that omits an *access_id_list* is permitted. A **PUBLIC** statement without an *access_id_list* sets the default accessibility to public for all potentially accessible entities in the module. If the module contains such a statement, it cannot also include a **PRIVATE** statement without an *access_id_list*. If the module does not contain a **PRIVATE** statement without an *access_id_list*, the default accessibility is public. Entities whose accessibility is not explicitly specified have default accessibility.

A procedure that has a generic identifier that is public is accessible through that identifier, even if its specific identifier is private. If a module procedure contains a private dummy argument or function result whose type has private accessibility, the module procedure must be declared to have private accessibility and must not have a generic identifier that has public accessibility.

IBM Although an entity with public accessibility cannot have the **STATIC** attribute, public entities in a module are unaffected by **IMPLICIT STATIC** statements in the module. **IBM**

Table 51. Attributes compatible with the PUBLIC attribute

ALLOCATABLE **1**	EXTERNAL	PROTECTED **1**
ASYNCHRONOUS	INTRINSIC	SAVE
CONTIGUOUS **2**	PARAMETER	TARGET
DIMENSION	POINTER	VOLATILE
Note: **1** Fortran 2003 **2** Fortran 2008		

Examples

```
MODULE MC
   PRIVATE                      ! Default accessibility declared as private
   PUBLIC GEN                   ! GEN declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1
   END INTERFACE
   CONTAINS
      SUBROUTINE SUB1(I)
         INTEGER I
         I = I + 1
      END SUBROUTINE SUB1
END MODULE MC
PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)                  ! SUB1 referenced because GEN has public
                                !    accessibility and appropriate argument
                                !    is passed
   PRINT *, K                   ! Value printed is 6
END PROGRAM
```

Related information

- "PRIVATE" on page 432
- "PROTECTED (Fortran 2003)" on page 438
- "Modules" on page 177

READ

Purpose

The **READ** statement is the data transfer input statement.

Syntax

```
►►──READ──┬─name─────────────────────────────────────────────────┬──►◄
          │                                                       │
          ├─format─┬──────────────────────┬──────────────────────┤
          │        └─,──input_item_list───┘                       │
          │                                                       │
          └─(──io_control_list──)─┬───────────────────────┬───────┘
                                  └──input_item_list──────┘
```
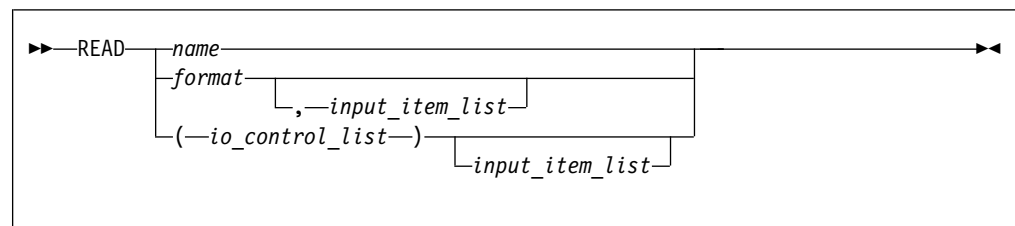
format   A format identifier that must not be a Hollerith constant. See **FMT=**_format_
      for more information.

name   A namelist group name.

input_item

An input list item. An input list specifies the data to be transferred. An
input list item can be:

- A variable name, but not for an assumed-size array. An array is treated
  as if all of its elements were specified in the order they are arranged in
  storage.

  A pointer must be associated with a definable target, and an allocatable
  object must be allocated. A derived-type object cannot have any ultimate
  component that is outside the scoping unit of this statement. The

evaluation of *input_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a formatted statement are treated as if they appear in the order of the derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).

- An implied-**DO** list, as described under "Implied-DO List" on page 448.

F2003 An *input_item* must not be a procedure pointer. **F2003**

io_control

is a list that must contain one unit specifier (**UNIT=**) and can also contain one of each of the other valid specifiers described below.

[UNIT=] *u*

is a unit specifier that specifies the unit to be used in the input 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 expression whose value can be in the range 1 through 2147483647
- **IBM** An asterisk, which identifies external unit 5 and is preconnected to standard input **IBM**
- **F2008** A NEWUNIT value **F2008**

An internal file identifier refers to an internal file. It is the name of a character variable that cannot be an array section with a vector subscript.

If the optional characters **UNIT=** are omitted, *u* must be the first item in *io_control_list*. If the optional characters **UNIT=** are specified, either the optional characters **FMT=** or the optional characters **NML=** must also be present.

[FMT=] *format*

is a format specifier that specifies the format to be used in the input operation. *format* is a format identifier that can be:

- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.
- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.

- A character constant. 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.
- A character variable 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 listed under "FORMAT" on page 376 can be used between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis. If *format* is an array element, the format identifier must not exceed the length of the array element.
- An array of noncharacter intrinsic type. The data must be a valid format identifier as described under character array.

- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.
- A namelist specifier that specifies a previously-defined namelist.

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 with the optional characters **UNIT=** omitted. Both **NML=** and **FMT=** cannot be specified in the same input statement.

ADVANCE= *char_expr*

is an advance specifier that determines whether nonadvancing input occurs for this statement. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. If **NO** is specified, nonadvancing input occurs. If **YES** is specified, advancing, formatted sequential or stream input occurs. The default value is **YES**. **ADVANCE=** can be specified only in a formatted sequential or formatted stream **READ** statement with an explicit format specification that does not specify an internal file unit specifier.

ASYNCH= *char_expr* **(IBM extension)**

is an asynchronous I/O specifier that indicates whether an explicitly connected unit is to be used for asynchronous I/O.

char_expr is a scalar character expression whose value is either **YES** or **NO**. **YES** specifies that asynchronous data transfer statements are permitted for this connection. **NO** specifies that asynchronous data transfer statements are not permitted for this connection. The value specified will be in the set of transfer methods permitted for the file. If this specifier is omitted, the default value is **NO**.

Preconnected units are connected with an **ASYNCH=** value of **NO**.

The **ASYNCH=** value of an implicitly connected unit is determined by the first data transfer statement performed on the unit. If the first statement performs an asynchronous data transfer and the file being implicitly connected permits asynchronous data transfers, the **ASYNCH=** value is **YES**. Otherwise, the **ASYNCH=** value is **NO**.

ASYNCHRONOUS= *char_expr* **(Fortran 2003)**

allows execution to continue without waiting for the data transfer to complete. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. **ASYNCHRONOUS=YES** must not appear unless **UNIT=** specifies a file unit number. If **ID=** appears, an **ASYNCHRONOUS=YES** must also appear.

A statement and the I/O operation are synchronous if **ASYNCHRONOUS=NO** or if both **ASYNCHRONOUS=** and **ID=** are absent. For **ASYNCHRONOUS=YES** or if **ID=** appears, asynchronous I/O is permitted only for external files opened with **ASYNCHRONOUS=YES** in the **OPEN** statement.

If a variable is used in an asynchronous data transfer statement as an item in an I/O list, a group object in a namelist or as a **SIZE=** specifier, the base object of the *data_ref* is implicitly given the **ASYNCHRONOUS** attribute in the scoping unit of the data transfer statement. For asynchronous nonadvancing input, the storage units specified in the **SIZE=** specifier become defined with the count of the characters transferred when the corresponding wait operation is executed. For asynchronous output, a pending I/O storage sequence affector shall not be redefined, become

undefined, or have its pointer association status changed. For asynchronous input, a pending I/O storage sequence affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the **VALUE** attribute, or have its pointer association status changed.

When an error, end-of-file or end-of-record condition occurs for a previously executed asynchronous data transfer statement , a wait operation is performed for all pending data transfer operations on that unit. When a condition occurs during a subsequent statement, any actions specified by **IOSTAT=**, **IOMSG=**, **ERR=**, **END=**, and **EOR=** specifiers for that statement are taken.

A wait operation is performed by a **WAIT**, **CLOSE**, or file positioning statement.

END= *stmt_label*
    is an end-of-file specifier that specifies a statement label at which the program is to continue if an endfile record is encountered and no error occurs. An external file is positioned after the endfile record; the **IOSTAT=** specifier, if present, is assigned a negative value; and the **NUM=** specifier, if present, is assigned an integer value. If an error occurs and the statement contains the **SIZE=** specifier, the specified variable becomes defined with an integer value. Coding the **END=** specifier suppresses the error message for end-of-file. This specifier can be specified for a unit connected for either sequential or direct access.

EOR= *stmt_label*
    is an end-of-record specifier. If the specifier is present, an end-of-record condition occurs, and no error condition occurs during execution of the statement. If **PAD=** exists, the following also occur:

1. If the **PAD=** specifier has the value **YES**, the record is padded with blanks to satisfy the input list item and the corresponding data edit descriptor that requires more characters than the record contains.
2. Execution of the **READ** statement terminates.
3. The file specified in the **READ** statement is positioned after the current record.
4. If the **IOSTAT=** specifier is present, the specified variable becomes defined with a negative value different from an end-of-file value.
5. If the **SIZE=** specifier is present, the specified variable becomes defined with an integer value.
6. Execution continues with the statement containing the statement label specified by the **EOR=** specifier.
7. End-of-record messages are suppressed.

BLANK= *char_expr* **(Fortran 2003)**
    controls the default interpretation of blanks when you are using a format specification. *char_expr* is a scalar character expression whose value, when any trailing blanks are removed, is either **NULL** or **ZERO**. If **BLANK=** is specified, you must use **FORM='FORMATTED'**. If **BLANK=** is not specified and you specify **FORM='FORMATTED'**, **NULL** is the default.

DECIMAL= *char_expr* **(Fortran 2003)**
    temporarily changes the default decimal edit mode for the duration of an I/O statement. *char_expr* is a scalar character expression whose value must

evaluate to either **POINT**, or **COMMA**. After each **READ** statement, the mode defaults to whatever decimal mode was specified (or assumed) on the **OPEN** statement for that unit.

ERR= *stmt_label*

is an error specifier that specifies the statement label of an executable statement to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

ID= *integer_variable* **(IBM extension)**

indicates that the data transfer is to be done asynchronously. The *integer_variable* is an integer variable. If no error is encountered, the *integer_variable* is defined with a value after executing the asynchronous data transfer statement. This value must be used in the matching **WAIT** statement.

> F2003 A child data transfer statement must not contain the **ID=** specifier. F2003

Asynchronous data transfer must either be direct unformatted, sequential unformatted or stream unformatted. Asynchronous I/O to internal files is prohibited. Asynchronous I/O to raw character devices (for example, to tapes or raw logical volumes) is prohibited. The *integer_variable* must not be associated with any entity in the data transfer I/O list, or with a *do_variable* of an *io_implied_do* in the data transfer I/O list. If the *integer_variable* is an array element reference, its subscript values must not be affected by the data transfer, the *io_implied_do* processing, or the definition or evaluation of any other specifier in the *io_control_spec*.

IOMSG= *iomsg_variable* **(Fortran 2003)**

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

is an input/output status specifier that specifies the status of the input/output operation. *ios* is an integer variable. Coding the **IOSTAT=** specifier suppresses error messages. When the statement finishes execution, *ios* is defined with:

- A zero value if no error condition, end-of-file condition, or end-of-record condition occurs.
- A positive value if an error occurs.
- A negative value if an end-of-file condition is encountered and no error occurs.
- A negative value that is different from the end-of-file value if an end-of-record condition occurs and no error condition or end-of-file condition occurs.

PAD= *char_expr* **(Fortran 2003)**

specifies if input records are padded with blanks. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. If the value is **YES**, a formatted input record is padded with blanks if an input list is specified and the format specification requires more data from a record than the

record contains. If **NO** is specified, the input list and format specification must not require more characters from a record than the record contains. The default value is **YES**. The **PAD=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during output of a formatted record.

IBM If the **-qxlf77** compiler option specifies the **noblankpad** suboption and the file is being connected for formatted direct input/output, the default value is **NO** when the **PAD=** specifier is omitted. IBM

[NML=] *name*

is a namelist specifier that specifies a previously-defined namelist. If the optional characters **NML=**are not specified, the namelist name must appear as the second parameter in the list and the first item must be the unit specifier with **UNIT=** omitted. 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 statement.

NUM= *integer_variable* **(IBM extension)**

is a number specifier that specifies the number of bytes of data transmitted between the I/O list and the file. *integer_variable* is an integer variable. The **NUM=** specifier is only permitted for unformatted output. Coding the **NUM** parameter suppresses the indication of an error that would occur if the number of bytes represented by the 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 output list items is not written into subsequent records.

POS=*integer_expr* **(Fortarn 2003)**

is an integer expression greater than 0. **POS=** specifies the file position of the file storage unit to be read in a file connected for stream access. You must not use this specifier for a file that cannot be positioned or in a child data transfer statement.

REC= *integer_expr*

is a record specifier that specifies the number of the record to be read. F2003 If the control information list contains a **REC=** specifier, the statement is a direct access input/output statement. You must not use this specifier in a child data transfer statement. F2003

integer_expr is an integer expression whose value is positive. A record specifier is not valid if list-directed or namelist formatting is used and if the unit specifier specifies an internal file. IBM The **END=** specifier can appear concurrently. IBM The record specifier represents the relative position of a record within a file. The relative position number of the first record is 1. You must not specify **REC=** in data transfer statements that specify a unit connected for stream access, or use the **POS=** specifier.

ROUND= *char-expr* **(Fortran 2003)**

temporarily changes the current value of the I/O rounding mode for the duration of this I/O statement. If omitted, then the rounding mode is unchanged. *char-expr* evaluates to either **UP**, **DOWN**, **ZERO**, **NEAREST**, **COMPATIBLE** or **PROCESSOR_DEFINED**

The rounding mode helps specify how decimal numbers are converted to an internal representation, (that is, in binary) from a character

representation and vice versa during formatted input and output. The rounding modes have the following functions:

- In the **UP** rounding mode the value from the conversion is the smallest value that is greater than or equal to the original value.
- In the **DOWN** rounding mode the value from the conversion is the greatest value that is smaller than or equal to the original value.
- In the **ZERO** rounding mode the value from the conversion is the closest value to the original value, and not greater in magnitude.
- In the **NEAREST** rounding mode the value from the conversion is the closer of the two nearest representable values. If both values are equally close then the even value will be chosen. In IEEE rounding conversions, NEAREST corresponds to the ieee_nearest rounding mode as specified by the IEEE standard.
- In the **COMPATIBLE** rounding mode the value from the conversion is the closest of the two nearest representable values, or the value further away from zero if halfway between.
- In the **PROCESSOR_DEFINED** rounding mode the value from the conversion is processor dependent and may correspond to the other modes. In XL Fortran, the **PROCESSOR_DEFINED** rounding mode will be the rounding mode you choose in the floating-point control register. If you do not set the floating-point control register explicitly, the default rounding mode is **NEAREST**.

SIZE= *count*

A character count specifier that determines how many characters are transferred by data edit descriptors during execution of the current input statement. *count* is an integer variable. Blanks that are inserted as padding are not included in the count.

Rules

Any statement label specified by the **ERR=**, **EOR=** and **END=** specifiers must refer to a branch target statement that appears in the same scoping unit as the **READ** statement.

If either the **EOR=** specifier or the **SIZE=** specifier is present, the **ADVANCE=** specifier must also be present and must have the value **NO**.

IBM If a **NUM=** specifier is present, neither a format specifier nor a namelist specifier can be present. **IBM**

Variables specified for the **IOSTAT=**, **SIZE=** and **NUM=** specifiers must not be associated with any input list item, namelist list item, or the **DO** variable of an implied-**DO** list. If such a specifier variable is an array element, its subscript values must not be affected by the data transfer, any implied-**DO** processing, or the definition or evaluation of any other specifier.

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.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered during a synchronous data transfer, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If the **ERR=** or **IOSTAT=** specifiers are set and an error is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If the **END=** or **IOSTAT=** specifiers are set and an end-of-file condition is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If a conversion error is encountered and the **CNVERR** run-time option is set to **NO**, **ERR=** is not branched to, although **IOSTAT=** may be set.

If **IOSTAT=** and **ERR=** are not specified,
- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.
- The program continues to the next statement when a conversion error is encountered if the **ERR_RECOVERY** run-time option is set to **YES**. If the **CNVERR** run-time option is set to **YES**, conversion errors are treated as recoverable errors; if **CNVERR=NO**, they are treated as conversion errors.

Examples

```
INTEGER A(100)
CHARACTER*4 B
READ *, A(LBOUND(A,1):UBOUND(A,1))
READ (7,FMT='(A3)',ADVANCE='NO',EOR=100) B
     ...
100 PRINT *, 'end of record reached'
END
```

Related information
- "Asynchronous Input/Output" on page 220
- *Implementation details of XL Fortran Input/Output* in the *XL Fortran Optimization and Programming Guide*
- "Conditions and IOSTAT values" on page 226
- "WRITE" on page 495
- "WAIT (Fortran 2003)" on page 491
- Chapter 9, "XL Fortran Input/Output," on page 215
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*
- "Deleted features" on page 1054

Implied-DO List

```
►►──(──do_object_list── , ──do_variable = arith_expr1, arith_expr2─────────►

►──┬──────────────┬──)──────────────────────────────────────────────►◄
   └──,──┬──────────┬─┘
         └─arith_expr3─┘
```

do_object
        is an output list item

do_variable
        is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, **and** *arith_expr3*
        are scalar numeric expressions

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. 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.

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.

REAL

Purpose

A **REAL** type declaration statement specifies the length and attributes of objects and functions of type real. Initial values can be assigned to objects.

Syntax

```
►►──REAL──┬────────────────┬──┬──::────────────────────┬──entity_decl_list──►◄
          └─kind_selector─┘  └──,──attr_spec_list──::──┘
```

where:

attr_spec
        is any of the following:

ALLOCATABLE **1**	**INTRINSIC**	**PUBLIC**
ASYNCHRONOUS	**OPTIONAL**	**SAVE**
AUTOMATIC **2**	**PARAMETER**	**STATIC** **2**
BIND **1**	**POINTER**	**TARGET**
DIMENSION (*array_spec*)	**PRIVATE**	**VALUE** **1**
EXTERNAL	**PROTECTED** **1**	**VOLATILE**

INTENT (*intent_spec*)		
Note: **1** Fortran 2003 **2** IBM extension		

kind_selector

```
►►─┬──(─┬────────────────┬──int_constant_expr─)─┬─────────────────►◄
   │    └─KIND── = ─┘                           │
   │                                  (1)       │
   └─ * ─int_literal_constant─────────────────┘

Notes:

1    IBM extension.
```

> **IBM** specifies the length of real entities: 4, 8 or 16 . *int_literal_constant* cannot specify a kind type parameter. **IBM**

attr_spec
> For detailed information on rules about a particular attribute, refer to the statement of the same name.

intent_spec
> is either **IN**, **OUT**, or **INOUT**

::   is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
> is a list of dimension bounds

entity_decl

```
►►─a─┬──────────────────────────┬─┬──────────────────┬──────────►
     │       (1)       (2)      │ └─(─array_spec─)─┘
     └─ * ──────len──────┘
     │                            (3)        (4)
     └─(─array_spec─)─ * ─────────len────────┘

►─┬──────────────────────────┬─────────────────────────────────►◄
  │                   (5)     │
  ├─/─initial_value_list─/────┤
  ├─ = ─constant_expr─────────┤
  └─ => ─NULL()──────────────┘

Notes:

1    IBM extension.

2    IBM extension.

3    IBM extension.

4    IBM extension.

5    IBM extension.
```
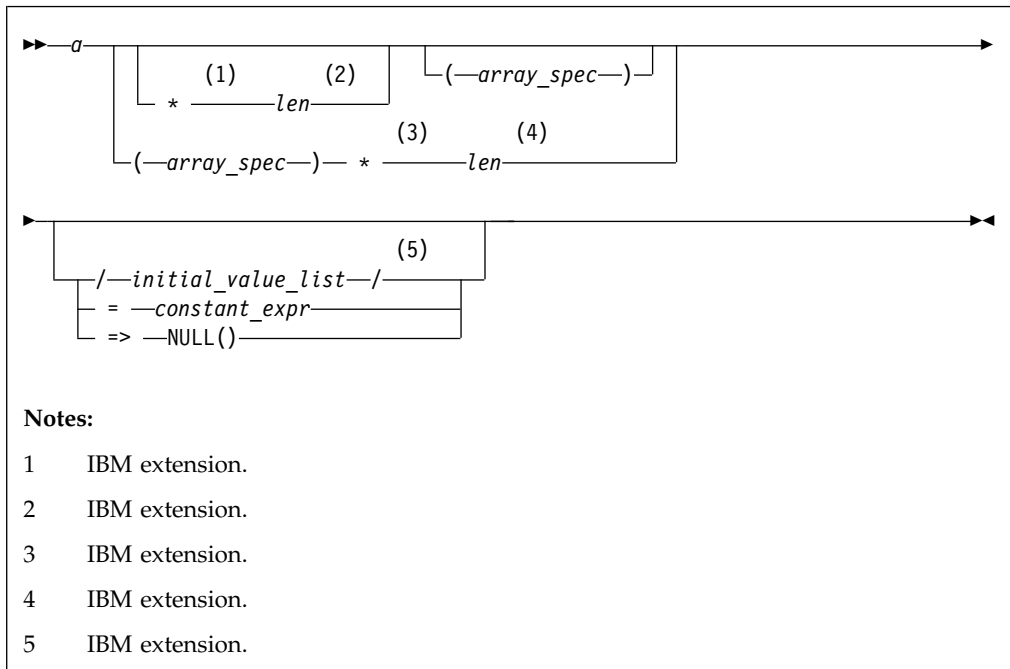
a
is an object name or function name. *array_spec* cannot be specified for a function name with an implicit interface.

len **(IBM extension)**
overrides the length as specified in *kind_selector*, and cannot specify a kind type parameter. The entity length must be an integer literal constant that represents one of the permissible length specifications.

initial_value **(IBM extension)**
provides an initial value for the entity specified by the immediately preceding name.

constant_expr
provides a constant expression for the entity specified by the immediately preceding name

=> **NULL()**
provides the initial value for the pointer object

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, an allocatable object, a function result, an object in a blank common block, an integer pointer, an external name, an intrinsic name, or an automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit.

▶IBM◀ The object also may be initialized if it appears in a named common block in a module or submodule. ▶IBM◀

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of => **NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined.

If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the `F2003` **ALLOCATABLE** or `F2003` **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

`IBM` If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. `IBM`

Examples
```
REAL(8), POINTER :: RPTR
REAL(8), TARGET  :: RTAR
```

Related information
- "Real" on page 38
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values

RECORD (IBM extension)

Purpose

The **RECORD** statement is a special form of type declaration statement. Unlike other type declaration statements, attributes for entities declared on the **RECORD** statement cannot be specified on the statement itself.

Syntax

```
           ┌──── , ─────────────────────────────────┐
           ▼                                         │
►►──RECORD───/ type_name /──────────record_obj_dcl_list──────►◄
                         └─ :: ─┘
```

record_obj_dcl:

```
►►──record_object_name──────────────────────►◄
                        └─(-array_spec-)─┘
```

record_stmt:

```
           ┌──── , ─────────────────────────────────┐
           ▼                                         │
►►──RECORD───/ type_name /──────────record_obj_dcl_list──────►◄
                         └─ :: ─┘
```

record_obj_dcl:

```
►►──record_object_name──────────────────────►◄
                        └─(-array_spec-)─┘
```
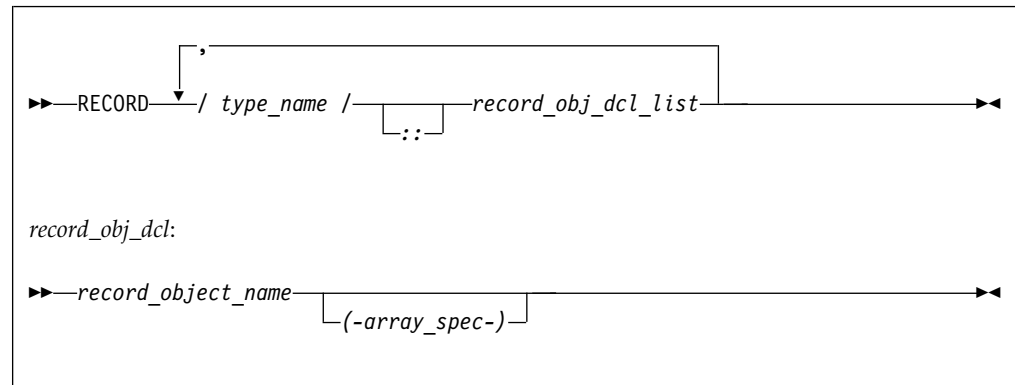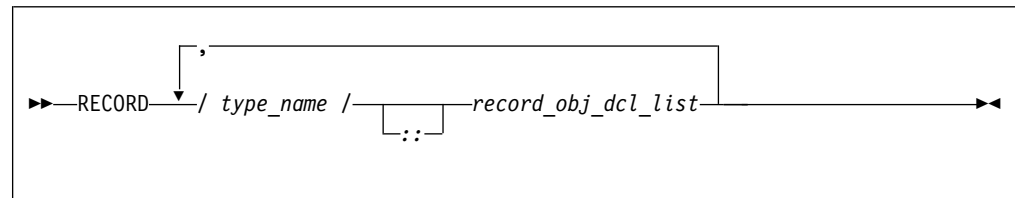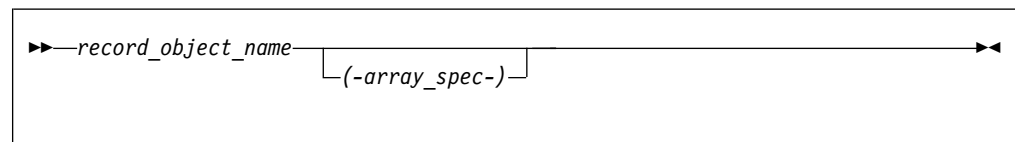
where *type_name* must be the name of a derived type that is accessible in the scoping unit.

Rules

Entities can not be initialized in a **RECORD** statement.

A *record_stmt* declares an entity to be of the derived type, specified by the *type_name* that most immediately precedes it.

The *RECORD* keyword must not appear as the *type_spec* of an **IMPLICIT** or **FUNCTION** statement.

A derived type with the **BIND** attribute must not be specified in a **RECORD** statement.

Chapter 11. Statements and attributes **453**

Examples

In the following example, a **RECORD** statement is used to declare a derived type variable.

```
STRUCTURE /S/
   INTEGER I
END STRUCTURE
STRUCTURE /DT/
   INTEGER I
END STRUCTURE
RECORD/DT/REC1,REC2,/S/REC3,REC4
```

Related information

- For further information on record structures and derived types, see Chapter 4, "Derived types," on page 49

RETURN

Purpose

The **RETURN** statement:
- In a function subprogram, ends the execution of the subprogram and returns control to the referencing statement. The value of the function is available to the referencing procedure.
- In a subroutine subprogram, ends the subprogram and transfers control to the first executable statement after the procedure reference or to an alternate return point, if one is specified.
- **IBM** In the main program, ends execution of the executable program. **IBM**

Syntax

```
►►──RETURN─────────────────────────────────────────────►◄
            │            (1)         │
            └─arith_expr─────────────┘

Notes:

1    Real or complex expressions are an IBM extension.
```
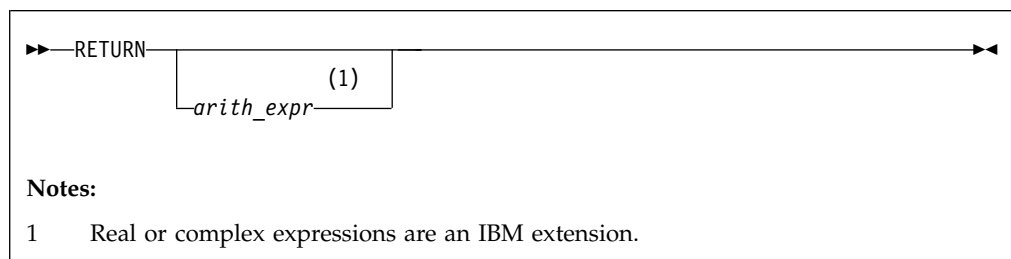
arith_expr
> A scalar integer, real, or complex expression. If the value of the expression is noninteger, it is converted to **INTEGER(4)** before use. *arith_expr* must not be a Hollerith constant.

Rules

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 $1 \leq m \leq$ 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 procedure 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, 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 procedure at the statement following the **CALL** statement.

Executing a **RETURN** statement terminates the association between the dummy arguments of the subprogram and the actual arguments supplied to that instance of the subprogram. All entities local to the subprogram become undefined, except as noted under "Events causing undefinition" on page 24.

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.

Examples

```
CALL SUB(A,B)
CONTAINS
  SUBROUTINE SUB(A,B)
    INTEGER :: A,B
    IF (A.LT.B)
      RETURN            ! Control returns to the calling procedure
    ELSE
      ...
    END IF
  END SUBROUTINE
END
```

Related information
- "Asterisks as dummy arguments" on page 206
- "Actual argument specification" on page 192 for a description of alternate return points
- "Events causing undefinition" on page 24

REWIND

Purpose

The **REWIND** statement positions an external file connected for sequential access at the beginning of the first record of the file. `F2003` For stream access, the **REWIND** statement positions a file at its initial point. `F2003`

`F2003` Execution of a **REWIND** statement performs a wait operation for any pending asynchronous data transfer operations for the specified unit. `F2003`

Syntax

```
>>--REWIND---u-----------------------------------------><
             |                    |
             |-(--position_list--)-|
```
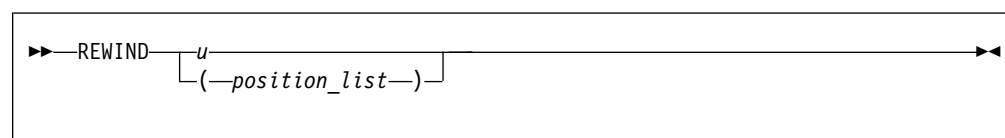
u   An external unit identifier that must not be an asterisk or a Hollerith
            constant.

position_list

> 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*

> 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 expression. The integer expression has one of the
> following values:
>
> • A value in the range 1 through 2147483647
> • ▌F2008▐ A NEWUNIT value ▌F2008▐
>
> If the optional characters **UNIT=** are omitted, *u* must be the first item in
> *position_list*.

ERR= *stmt_label*

> An error specifier that specifies the statement label of an executable
> statement in the same scoping unit to which control is to transfer in the
> case of an error. Coding the **ERR=** specifier suppresses error messages.

IOMSG= *iomsg_variable* **(Fortran 2003)**

> An input/output status specifier that specifies the message returned by the
> input/output operation. *iomsg_variable* is a scalar default character variable.
> It must not be a use-associated nonpointer protected variable. When the
> input/output statement containing this specifier finishes execution,
> *iomsg_variable* is defined as follows:
>
> • If an error, end-of-file, or end-of-record condition occurs, the variable is
>   assigned an explanatory message as if by assignment.
> • If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

> An input/output status specifier for the status of the input/output
> operation. *ios* is a scalar integer variable. When the **REWIND** statement
> finishes executing, *ios* is defined with:
> • A zero value if no error condition occurs
> • A positive value if an error occurs.

Rules

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 exists, 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** statement has no effect. The
REWIND statement causes a subsequent **READ** or **WRITE** statement referring to *u*
to read data from or write data to the first record of the external file associated
with *u*.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered, transfer is
made to the statement specified by the **ERR=** specifier and a positive integer value
is assigned to *ios*.

▌IBM▐ If **IOSTAT=** and **ERR=** are not specified,
• the program stops if a severe error is encountered.

- the program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.

◀ **IBM**

Examples

```
REWIND (9, IOSTAT=IOSS)
```

Related information
- "Conditions and IOSTAT values" on page 226
- Chapter 9, "XL Fortran Input/Output," on page 215
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*

SAVE

Purpose

The **SAVE** attribute specifies the names of objects and named common blocks whose definition status you want to retain after control returns from the subprogram where you define the variables and named common blocks.

Syntax

```
▶▶──SAVE──────────────────────────────────────────────────────────▶◀
              ┌──────────────,◀─────────────┐
              ▼                              │
          ┌─────────┐   ──object_name────────┤
          └──::─────┘   └─/──common_block_name──/─┘
```
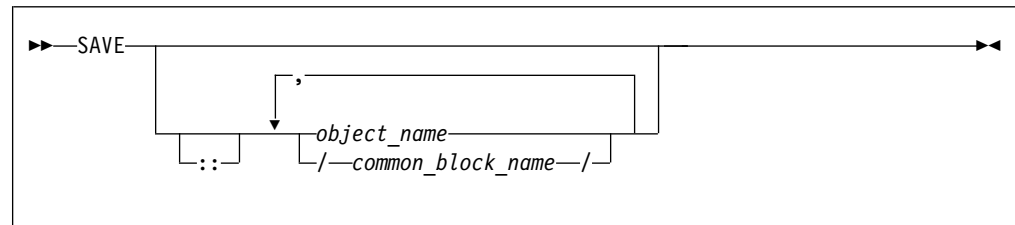
Rules

A **SAVE** statement without a list is treated as though it contains the names of all common items and local variables in the scoping unit. A common block name having the **SAVE** attribute has the effect of specifying all the entities in that named common block.

Within a function or subroutine subprogram, a variable whose name you specify with the **SAVE** attribute does not become undefined as a result of a **RETURN** or **END** statement in the subprogram.

object_name cannot be the name of a dummy argument, pointee, procedure, automatic object, or common block entity.

If a local entity specified with the **SAVE** attribute (and not in a common block) is in a defined state at the time that a **RETURN** or **END** statement is encountered in a subprogram, that entity is defined with the same value at the next reference of that subprogram. Saved objects are shared by all instances of the subprogram.

▶**F2008** You can also specify the **SAVE** attribute in a **BLOCK** construct.◀**F2008**

XL Fortran permits function results to have the **SAVE** attribute. To indicate that a function result is to have the **SAVE** attribute, the function result name must be explicitly specified with the **SAVE** attribute. That is, a **SAVE** statement without a list does not provide the **SAVE** attribute for the function result.

Variables declared as **SAVE** are shared amongst threads. To thread-safe an application that contains shared variables, you must either serialize access to the static data using locks, or make the data thread-specific. One method of making the data thread-specific is to move the static data into a named **COMMON** block that has been declared **THREADLOCAL**. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See Pthreads library module in the *XL Fortran Optimization and Programming Guide* for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See **CRITICAL**/**END CRITICAL** in the *XL Fortran Optimization and Programming Guide* for more information. The **THREADLOCAL** directive ensures that common blocks are local to each thread. See **THREADLOCAL** in the *XL Fortran Optimization and Programming Guide* for more information.

─────────── **End of IBM extension** ───────────

Table 52. Attributes compatible with the SAVE attribute

ALLOCATABLE **1**	POINTER	STATIC **3**
ASYNCHRONOUS	PRIVATE	TARGET
CONTIGUOUS **2**	PROTECTED **1**	VOLATILE
DIMENSION	PUBLIC	
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, SAVE :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
    ENDIF
    PRINT *, J                     ! Output on first call is 2
                                   ! Output on second call is 3
  END SUBROUTINE
END
```

Related information
- "COMMON" on page 315
- **THREADLOCAL** in the *XL Fortran Optimization and Programming Guide*
- "Definition status of variables" on page 21
- "Storage classes for variables (IBM extension)" on page 28

SELECT CASE

Purpose

The **SELECT CASE** statement is the first statement of a **CASE** construct. It provides a concise syntax for selecting, at most, one of a number of statement blocks for execution.

Syntax

```
►►─┬─────────────────────┬─SELECT CASE─(─case_expr─)──────────────►◄
   └─case_construct_name─:─┘
```
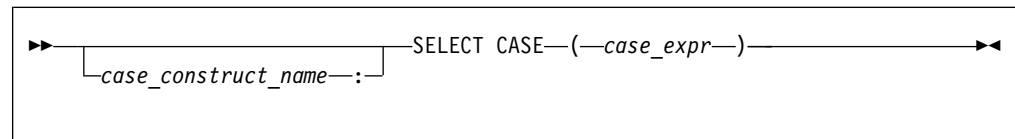
case_construct_name
        A name that identifies the **CASE** construct

case_expr
        A scalar expression of type integer, character or logical

Rules

When a **SELECT CASE** statement is executed, the *case_expr* is evaluated. The resulting value is called the case index, which is used for evaluating control flow within the case construct.

If the *case_construct_name* is specified, it must appear on the **END CASE** statement and optionally on any **CASE** statements within the construct.

IBM The *case_expr* must not be a typeless constant or a **BYTE** data object. **IBM**

Examples

```
ZERO: SELECT CASE(N)          ! start of CASE construct ZERO

   CASE DEFAULT ZERO
        OTHER: SELECT CASE(N) ! start of CASE construct OTHER
          CASE(:-1)
              SIGNUM = -1
          CASE(1:) OTHER
              SIGNUM = 1
        END SELECT OTHER
   CASE (0)
     SIGNUM = 0

END SELECT ZERO
```

Related information
- "CASE construct" on page 145
- "CASE" on page 305
- "END (Construct)" on page 351, for details on the **END SELECT** statement

SELECT TYPE (Fortran 2003)

Purpose

The **SELECT TYPE** statement is the first statement in a **SELECT TYPE** construct. The construct can have any number of statement blocks, only one of which is selected for execution. The selection is based on the dynamic type and the **KIND TYPE** parameters of an expression, which you specify in the *selector*, the type and the corresponding **KIND TYPE** parameters of each type guard statement.

Syntax

```
►►─┬──────────────────────────┬─SELECT TYPE─(─┬─────────────────────┬─selector─)──────►◄
   └─select_construct_name─:──┘               └─associate_name─=>───┘
```
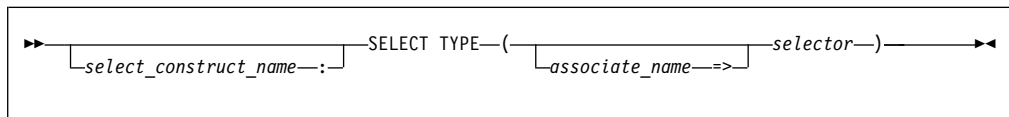
select_construct_name
> A name that identifies the **SELECT TYPE** construct.

associate_name
> A name that is associated with the *selector* when executing the **SELECT TYPE** statement.

selector An expression, evaluated when executing the **SELECT TYPE** statement. The result must be polymorphic.

Rules

If the *selector* is not a named variable, *associate_name* must appear. If the *selector* is not a definable variable or is a variable that has a vector subscript, *associate_name* must not appear in a variable definition context.

The selector must be polymorphic.

If the *select_construct_name* is specified, it must appear on the *END SELECT* statement and optionally on any type guard statements within the construct.

Related information
- "SELECT TYPE construct (Fortran 2003)" on page 147
- "Type Guard (Fortran 2003)" on page 483
- "END (Construct)" on page 351, for details on the **END SELECT** statement

SEQUENCE

Purpose

The **SEQUENCE** statement specifies that the order of the components in a derived-type definition establishes the storage sequence for objects of that type. Such a type becomes a *sequence derived type*.

Syntax

```
►►──SEQUENCE────────────────────────────────────────────────►◄
```

Rules

The **SEQUENCE** statement can be specified only once in a derived-type definition.

If a component of a sequence derived type is of derived type, that derived type must also be a sequence derived type.

IBM The size of a sequence derived type is equal to the number of bytes of storage needed to hold all of the components of that derived type. **IBM**

Use of sequence derived types can lead to misaligned data, which can adversely affect the performance of a program.

F2003 If a derived type definition has procedures or the **BIND** attribute, the **SEQUENCE** statement cannot be specified. Also, **SEQUENCE** cannot be specified for an extended type. **F2003**

Examples

```
TYPE PERSON
  SEQUENCE
  CHARACTER*1 GENDER     ! Offset 0
  INTEGER(4) AGE         ! Offset 1
  CHARACTER(30) NAME     ! Offset 5
END TYPE PERSON
```

Related information

- Chapter 4, "Derived types," on page 49
- "TYPE" on page 333
- "END TYPE" on page 355

Statement Function

Purpose

A statement function defines a function in a single statement.

Syntax

```
►►──name──(──┬──────────────────────┬──)── = ──scalar_expression──────►◄
             └─dummy_argument_list─┘
```

name   is the name of the statement function. It must not be supplied as a procedure argument and cannot be the target of a procedure pointer.

dummy_argument
    can only appear once in the dummy argument list of any statement

function. The dummy arguments have the scope of the statement function statement, and the same types and type parameters as the entities of the same names in the scoping unit containing the statement function.

Rules

A statement function is local to the scoping unit in which it is defined. It must not be defined in the scope of a module `F2008` or submodule `F2008` .

name determines the data type of the value returned from the statement function. If the data type of *name* does not match that of the scalar expression, the value of the scalar expression is converted to the type of *name* in accordance with the rules for assignment statements.

The names of the function and all the dummy arguments must be specified, explicitly or implicitly, to be scalar data objects.

The scalar expression can be composed of constants, references to variables, references to functions and function dummy procedures, and intrinsic operations. If the expression contains a reference to a function or function dummy procedure, the reference must not require an explicit interface, the function must not require an explicit interface or be a transformational intrinsic, and the result must be scalar. If an argument to a function or function dummy procedure is array-valued, it must be an array name.

`IBM` With XL Fortran, the scalar expression can also reference a structure constructor. `IBM`

The scalar expression can reference another statement function that is either:
* Declared previously in the same scoping unit, or
* Declared in the host scoping unit.

Named constants and arrays whose elements are referenced in the expression must be declared earlier in the scoping unit or be made accessible by use or host association.

Variables that are referenced in the expression must be either:
* Dummy arguments of the statement function, or
* Accessible in the scoping unit

If an entity in the expression is typed by the implicit typing rules, its type must agree with the type and type parameters given in any subsequent type declaration statement.

An external function reference in the scalar expression must not cause any dummy arguments of the statement function to become undefined or redefined.

If the statement function is defined in an internal subprogram and if it has the same name as an accessible entity from the host, precede the statement function definition with an explicit declaration of the statement function name. For example, use a type declaration statement.

The length specification for a statement function of type character or a statement function dummy argument of type character must be a constant specification expression.

Examples

```
PARAMETER (PI = 3.14159)
REAL AREA,CIRCUM,R,RADIUS
AREA(R) = PI * (R**2)             ! Define statement functions
CIRCUM(R) = 2 * PI * R            !   AREA and CIRCUM

! Reference the statement functions
PRINT *,'The area is: ',AREA(RADIUS)
PRINT *,'The circumference is: ',CIRCUM(RADIUS)
```

Related information

- "Dummy arguments" on page 194
- "Function reference" on page 186
- "Determining Type" on page 18, for information on how the type of the statement function is determined

STATIC (IBM extension)

Purpose

The **STATIC** attribute specifies that a variable has a storage class of static; that is, the variable remains in memory for the duration of the program and its value is retained between calls to the procedure.

Syntax
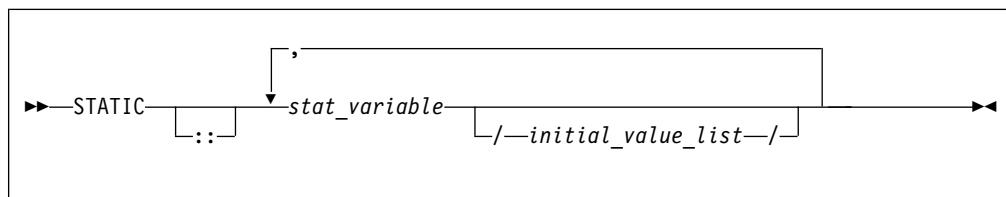
stat_variable
> is a variable name or an array declarator that can specify an *explicit_shape_spec_list* or a *deferred_shape_spec_list*.

initial_value
> provides an initial value for the variable specified by the immediately preceding name. Initialization occurs as described in "DATA" on page 327.

Rules

If *stat_variable* is a result variable, it must not be of type character or of derived type. Dummy arguments, automatic objects and pointees must not have the **STATIC** attribute. A variable that is explicitly declared with the **STATIC** attribute cannot be a common block item.

A variable must not have the **STATIC** attribute specified more than once in the same scoping unit.

Local variables have a default storage class of automatic. See the **-qsave** option in the *XL Fortran Compiler Reference* for details on the default settings with regard to the invocation commands.

Variables declared as **STATIC** are shared amongst threads. To thread-safe an application that contains shared variables, you must either serialize access to the static data using locks, or make the data thread-specific. One method of making the data thread-specific is to move the static data into a **COMMON** block that has been declared **THREADLOCAL**. The **Pthreads** library module provides mutexes to allow you to serialize access to the data using locks. See **Pthreads library module** in the *XL Fortran Optimization and Programming Guide* for more information. The *lock_name* attribute on the **CRITICAL** directive also provides the ability to serialize access to data. See **CRITICAL**/**END CRITICAL** in the *XL Fortran Optimization and Programming Guide* for more information. The **THREADLOCAL** directive ensures that common blocks are local to each thread. See **THREADLOCAL** in the *XL Fortran Optimization and Programming Guide* for more information.

Table 53. Attributes compatible with the STATIC attribute

ALLOCATABLE **1**	POINTER	SAVE
ASYNCHRONOUS	PRIVATE	TARGET
CONTIGUOUS **2**	PROTECTED **1**	VOLATILE
DIMENSION		
Note: **1** Fortran 2003 **2** Fortran 2008		

Examples

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, STATIC :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
    ENDIF
    PRINT *, J                    ! Output on first call is 2
                                  ! Output on second call is 3
  END SUBROUTINE
END
```

Related information
* "Storage classes for variables (IBM extension)" on page 28
* "COMMON" on page 315
* **THREADLOCAL** in the *XL Fortran Optimization and Programming Guide*

STOP

Purpose

The **STOP** statement initiates normal termination, which terminates the execution of the program. If a *stop_code* is specified, the keyword "STOP" followed by the *stop_code* is printed to ERROR_UNIT.

Syntax

```
>>--STOP----------------------------------------------------><
         |_stop_code_|
```

where *stop_code* is:

```
                                  (1)
>>---+-scalar_char_constant_expr-+--------------------------------------><
     |                        (2) |
     |_scalar_int_constant_expr___|
```

Notes:

1   Fortran 2008
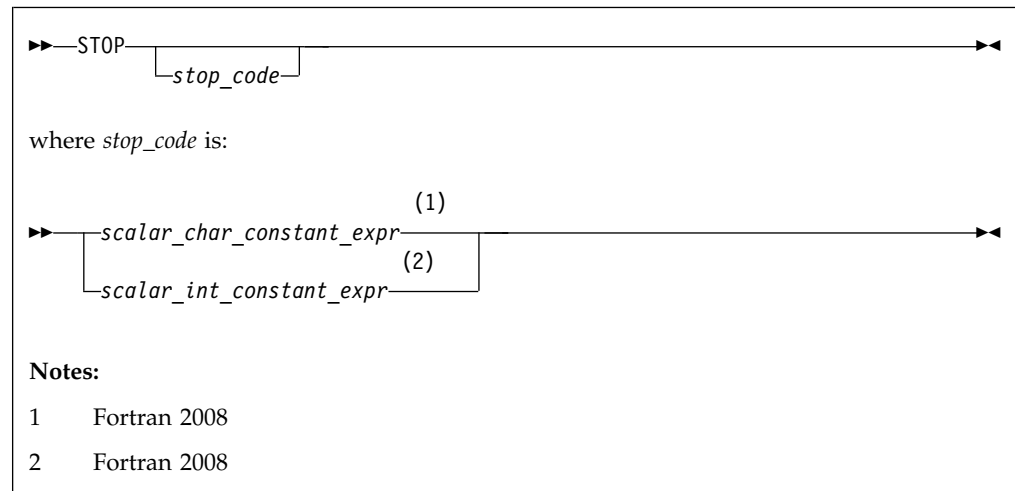
2   Fortran 2008

scalar_char_constant_expr
        is a scalar character constant expression

scalar_int_constant_expr
        is a scalar integer constant expression

Rules

IBM When a STOP statement is executed, a system return code is supplied and a message is printed to ERROR_UNIT, depending on whether the *stop_code* is specified:

- If the *stop_code* is *scalar_char_constant_expr*, the system return code is 0. The keyword "STOP" followed by the *stop_code* is printed.
- If the *stop_code* is *scalar_int_constant_expr*, XL Fortran sets the system return code to **MOD** (*stop_code*, 256). The keyword "STOP" followed by the *stop_code* is printed.
- If nothing is specified, the system return code is 0. No error message is printed.

The system return code is available in the Korn shell command variable *$?*.**IBM**

F2003 If you compile your program with **-qxlf2003=stopexcept**, floating-point exceptions that are signaling are displayed when the **STOP** statement is reached.
F2003

A **STOP** statement cannot terminate the range of a **DO** or **DO WHILE** construct.

Examples

The following example shows how the STOP statement works when different kinds of *stop_code* are specified.

```
INTEGER :: matrix(10, 10)
INTEGER :: op
INTEGER :: result_matrix(10, 10)
INTEGER, PARAMETER :: init_error = 10
```

```
                  INTEGER, PARAMETER :: process = 20
                  CHARACTER(LEN = 10), PARAMETER :: message = "Terminated"

                  matrix = 10
                  result_matrix = 10

                  ! If the initialization is wrong, the message "STOP 11" is printed.
                  ! The system return code is 11.

                  IF (ANY(result_matrix .NE. 10)) STOP init_error + 1

                  ! If the initialization is wrong, the message "STOP 12" is printed.
                  ! The system return code is 12.

                  IF (ANY(matrix .NE. 10)) STOP 12

                  op = WHICH_OP()

                  IF (op .LT. 1) THEN
                    ! If OP is less than 1, the message "STOP Program Terminated" is printed.
                    ! The system return code is 0.
                    STOP "Program " // message
                  ELSE IF (OP .EQ. 1) THEN
                    result_matrix = result_matrix + matrix

                    ! The message "STOP 21" is printed.
                    ! The system return code is 21.

                    STOP PROCESS + 1
                  ELSE IF (OP .EQ. 2) THEN
                    result_matrix = result_matrix - matrix

                    ! The message "STOP 22" is printed.
                    ! The system return code is 22.

                    STOP process + 2
                  ELSE
                    ! No message is printed.
                    ! The system return code is 0.
                    STOP
                  END IF
```

Related information

- F2008 ERROR STOP F2008

SUBMODULE (Fortran 2008)

Purpose

The **SUBMODULE** statement is the first statement of a submodule program unit.

Syntax

```
►►──SUBMODULE── (──parent_identifier──)──submodule_name──────────────────►◄


parent_identifier

►►──ancestor_module_name──────────────────────────────────────────────►◄
                        └──:──parent_submodule_name──┘
```
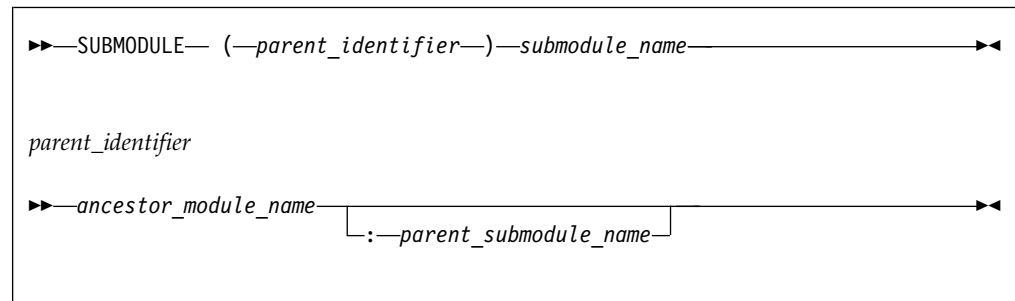
submodule_name
> Name of the submodule.

ancestor_module_name
> Name of a nonintrinsic module.

parent_submodule_name
> Name of a descendant submodule of the *ancestor_module*.

Rules

A submodule name cannot be accessed by use association; that is, it cannot be referred to by the **USE** statement.

A submodule cannot have the same name as any other program unit, external procedure, common block, or entity with a binding label in the program. In addition, the submodule name cannot be the same as any local name in the submodule. The name of a submodule can be the same as the name of another submodule if they do not have the same ancestor module.

If the **END** statement that completes the submodule specifies a submodule name, the name must be the same as the one that the **SUBMODULE** statement specifies.

Examples

```
MODULE mod
  ...
END MODULE

SUBMODULE (mod) mod1       ! parent_submodule_name is not specified
  ...
END SUBMODULE

SUBMODULE (mod) mod2       ! parent_submodule_name is not specified
  ...
END SUBMODULE

SUBMODULE (mod:mod1) mod3  ! parent_submodule_name is specified
  ...
END SUBMODULE
```

Related statements

- "END" on page 350
- "MODULE" on page 412
- "USE" on page 483
- "PRIVATE" on page 432

- "PROTECTED (Fortran 2003)" on page 438
- "PUBLIC" on page 440

Related information
- "Modules" on page 177
- "Submodules (Fortran 2008)" on page 181
- "Use association" on page 158

SUBROUTINE

Purpose

recursion

The **SUBROUTINE** statement is the first statement of a subroutine subprogram.

Syntax

Notes:

1   Fortran 2003
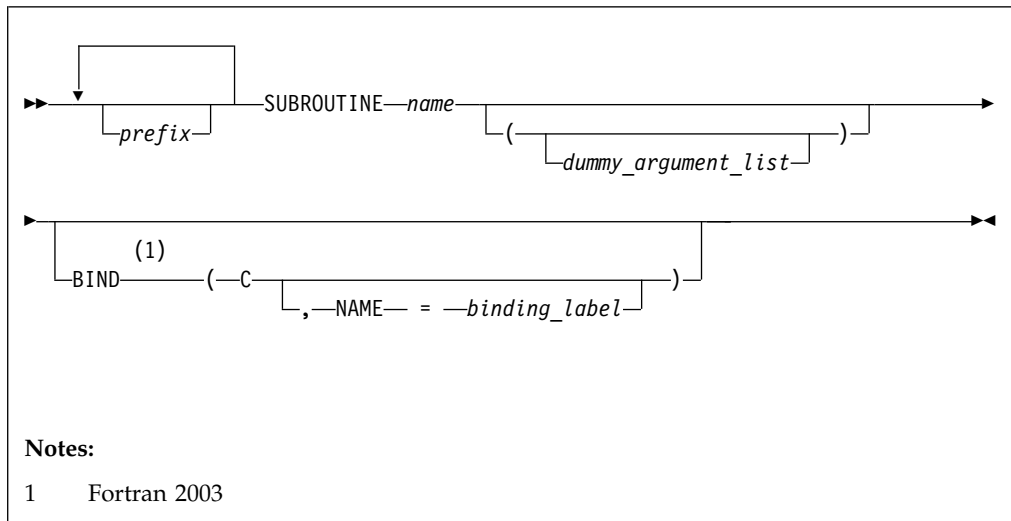
prefix   is one of the following:
- **ELEMENTAL**
- `F2008` **IMPURE** `F2008`
- `F2008` **MODULE** `F2008`
- **PURE**
- **RECURSIVE**

    Note: *type_spec* is not permitted as a prefix in a subroutine.

name   The name of the subroutine subprogram.

`F2003`

binding_label A scalar character constant expression. `F2003`

Rules

At most one of each kind of *prefix* can be specified. You cannot specify both the **RECURSIVE** and **ELEMENTAL** prefix specifiers. `F2008` You cannot specify both the **PURE** and **IMPURE** prefix specifiers. `F2008`

The subroutine name cannot appear in any other statement in the scope of the subroutine, unless recursion has been specified.

The **RECURSIVE** keyword must be specified if, directly or indirectly,
- The subroutine invokes itself.
- The subroutine invokes a procedure defined by an **ENTRY** statement in the same subprogram.
- An entry procedure in the same subprogram invokes itself.
- An entry procedure in the same subprogram invokes another entry procedure in the same subprogram.
- An entry procedure in the same subprogram invokes the subprogram defined by the **SUBROUTINE** statement.

If the **RECURSIVE** keyword is specified, the procedure interface is explicit within the subprogram.

Using the **PURE** or **ELEMENTAL** prefix indicates that the subroutine may be invoked by the compiler in any order as it is free of side effects. For elemental procedures, the keyword **ELEMENTAL** must be specified. If the **ELEMENTAL** keyword is specified, the **RECURSIVE** keyword cannot be specified.

`IBM` You can also call external procedures recursively when you specify the **-qrecur** compiler option, although XL Fortran disregards this option if the **SUBROUTINE** statement specifies the **RECURSIVE** keyword. `IBM`

`F2003` The **BIND** keyword implicitly or explicitly defines a binding label by which a procedure is accessed from the C programming language. A dummy argument cannot be zero-sized. A dummy argument for a procedure with the **BIND** attribute must have interoperable types and type parameters, and cannot have the **ALLOCATABLE** or **POINTER** attribute.

If the **SUBROUTINE** statement appears as part of an interface body that describes a dummy procedure, the **NAME=** specifier must not appear. An elemental procedure cannot have the **BIND** attribute. `F2003`

`F2008`

You can specify the **MODULE** prefix specifier for the **SUBROUTINE** statement of a module subprogram or of a nonabstract interface body that is declared in the scoping unit of a module or submodule. See Example 2.
- When you specify the **MODULE** prefix specifier for the **SUBROUTINE** statement of a module subprogram, the module subprogram is a separate module procedure.
- When you specify the **MODULE** prefix specifier for the **SUBROUTINE** statement of a nonabstract interface body, the interface body is a module procedure interface body.

The **BIND** attribute with the **NAME=** specifier is not allowed on an internal procedure.

F2008

Example 1

```
RECURSIVE SUBROUTINE SUB(X,Y)
  INTEGER X,Y
  IF (X.LT.Y) THEN
    RETURN
  ELSE
    CALL SUB(X,Y+1)
  END IF
END SUBROUTINE SUB
```

Example 2 (Fortran 2008)

```
MODULE m
  ! The MODULE prefix specifier is specified for the SUBROUTINE
  ! statement of a module procedure interface body.
  INTERFACE
    MODULE SUBROUTINE sub(arg)
      INTEGER :: arg
    END SUBROUTINE
  END INTERFACE
END MODULE

SUBMODULE (m) n

  CONTAINS
    ! The MODULE prefix specifier is specified for the SUBROUTINE
    ! statement of a separate module procedure.
    MODULE SUBROUTINE sub(arg)
      INTEGER :: arg
      arg = 1
    END SUBROUTINE
END SUBMODULE
```

Related information

- "Function and subroutine subprograms" on page 184
- "Dummy arguments" on page 194
- "Recursion" on page 208
- "CALL" on page 303
- "ENTRY" on page 358
- "Statement Function" on page 461
- "BIND (Fortran 2003)" on page 298
- "RETURN" on page 454
- "Definition status of variables" on page 21
- "Pure procedures" on page 208
- **-qrecur** option in the *XL Fortran Compiler Reference*
- "Interoperability of procedures" on page 888
- "Modules" on page 177
- "Submodules (Fortran 2008)" on page 181
- "Separate module procedures (Fortran 2008)" on page 188

TARGET

Purpose

The **TARGET** statement specifies the **TARGET** attribute of an entity. An object with the **TARGET** attribute may have a pointer associated with it.

Syntax

```
>>--TARGET--+------+--+->--variable_name--+-------------------+--+-->><
            |      |  |                   |  (--array_spec--)  |  |
            +--::--+  |                   +-------------------+  |
                      |                                          |
                      +------------------,-----------------------+
```
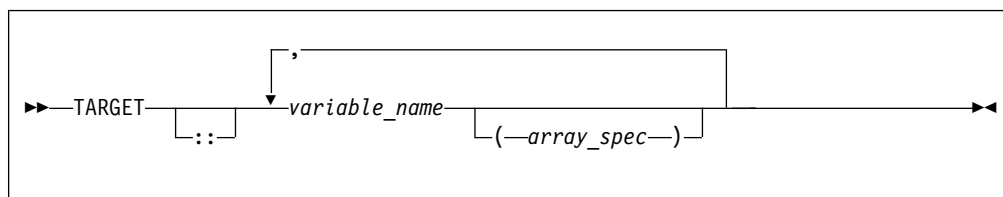
Rules

- If a data object has the **TARGET** attribute, then all of the data object's nonpointer subobjects will also have the **TARGET** attribute.
- A data object that does not have the **TARGET** attribute cannot be associated with an accessible pointer.
- A target cannot appear in an **EQUIVALENCE** statement.
- **IBM** A target cannot be an integer pointer or a pointee. **IBM**

Table 54. Attributes compatible with the TARGET attribute

ALLOCATABLE **1**	INTENT	SAVE
ASYNCHRONOUS	OPTIONAL	STATIC **3**
AUTOMATIC **3**	PRIVATE	VALUE **1**
DIMENSION	PROTECTED **1**	VOLATILE
CONTIGUOUS **2**	PUBLIC	
Notes: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

```
REAL, POINTER :: A,B
REAL, TARGET  :: C = 3.14
B => C
A => B        ! A points to C
```

Related information

- "POINTER (Fortran 90)" on page 426
- "ALLOCATED(X)" on page 580
- "DEALLOCATE" on page 331
- "Data pointer assignment" on page 130
- "Pointer association" on page 159

TYPE

Purpose

A **TYPE** type declaration statement specifies the type, type parameters, and attributes of objects and functions of derived type. Initial values can be assigned to objects.

F2008 The **TYPE** type declaration statement can declare entities of both derived type and intrinsic type. F2008

Syntax

```
>>--TYPE--(----intrinsic_type_spec----(1)----)--------------------->
                derived_type_spec
                                            --::--
                                            ,--attr_spec_list--::--

>--entity_decl_list-------------------------------------------><


Notes:

1      Fortran 2008
```
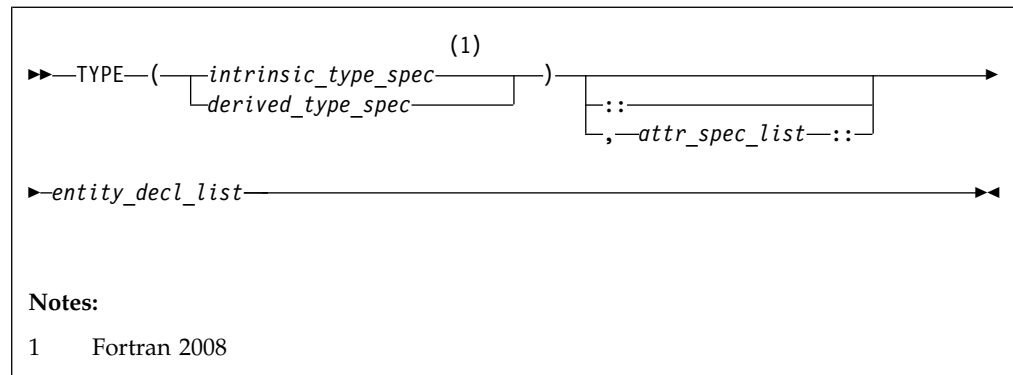
where:

F2008 *intrinsic_type_spec* F2008
> is the name of an intrinsic data type. For more information, see Chapter 3, "Intrinsic data types," on page 37.

derived_type_spec
> is the name of an extensible derived type. For more information, see "Type Declaration" on page 476.
>
> F2003 The derived type must not be abstract. F2003

::   is the double colon separator. It is required if attributes are specified, = *constant_expr* is used, or =>**NULL()** appears as part of any *entity_decl*.

attr_spec
> is any of the following attributes. For detailed information on rules about a particular attribute, refer to the statement of the same name.

ALLOCATABLE [1]	**INTRINSIC**	**PUBLIC**
ASYNCHRONOUS	**OPTIONAL**	**SAVE**
AUTOMATIC [2]	**PARAMETER**	**STATIC** [2]
BIND [1]	**POINTER**	**TARGET**
DIMENSION (*array_spec*)	**PRIVATE**	**VALUE** [1]
EXTERNAL	**PROTECTED** [1]	**VOLATILE**
INTENT (*intent_spec*)		
Note: [1] Fortran 2003 [2] IBM extension		

where:

array_spec
>   is a list of dimension bounds.

intent_spec
>   is one of **IN**, **OUT**, or **INOUT**.

entity_decl
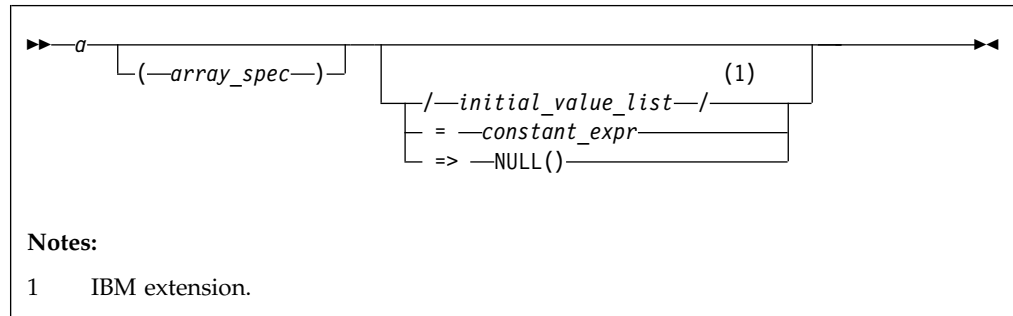
```
►►──a──────────────────────────────────────────────────────────────────►◄
        └─(─array_spec─)─┘ ┌────────────────────────(1)──┐
                           ├─/─initial_value_list─/───────┤
                           ├─ = ─constant_expr────────────┤
                           └─ => ──NULL()─────────────────┘

Notes:
1   IBM extension.
```

where:

a
>   is an object name or function name. *array_spec* cannot be specified
>   for a function with an implicit interface.

IBM *initial_value* **IBM**
>   provides an initial value for the entity specified by the immediately
>   preceding name. Initialization occurs as described in "DATA" on
>   page 327.

constant_expr
>   provides a constant expression for the entity specified by the
>   immediately preceding name.

=> **NULL()**
>   provides the initial value for a pointer object.

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear
  in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear
  in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type
  definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER**
attribute.

Entities in type declaration statements are constrained by the rules of any
attributes specified for the entities, as detailed in the corresponding attribute
statements.

Once a derived type has been defined, you can use it to define your data items using the **TYPE** type declaration statement. When an entity is explicitly declared to be of a derived type, that derived type must have been previously defined in the scoping unit or is accessible by use or host association.

The data object becomes an *object of derived type* or a *structure*. Each *structure component* is a subobject of the object of derived type.

If you specify the **DIMENSION** attribute, you are creating an array whose elements have a data type of that derived type.

Other than in specification statements, you can use objects of derived type as actual and dummy arguments, and they can also appear as items in input/output lists (unless the object has a component with the **POINTER** attribute), assignment statements, structure constructors, and the right side of a statement function definition. If a structure component is not accessible, a derived-type object cannot be used in an input/output list or as a structure constructor.

Objects of nonsequence derived type cannot be used as data items in **EQUIVALENCE** and **COMMON** statements. Objects of nonsequence data types cannot be integer pointees.

A nonsequence derived-type dummy argument must specify a derived type that is accessible through use or host association to ensure that the same derived-type definition defines both the actual and dummy arguments.

The type declaration statement overrides the implicit type rules in effect.

An object cannot be initialized in a type declaration statement if it is a dummy argument, allocatable object, function result, object in a blank common block, integer pointer, external name, intrinsic name, or automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit IBM or if it appears in a named common block in a module or submodule IBM .

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined.

If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification

statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in the *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the **ALLOCTABLE** or **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function. The derived type can be specified on the **FUNCTION** statement, provided the derived type is defined within the body of the function or is accessible via host or use association.

IBM If `T` or `F`, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. **IBM**

Example 1

The following code defines a derived type `people` using the **TYPE** type declaration statement.

```
TYPE people
  INTEGER age
  CHARACTER*20 name
END TYPE people
```

The following statement declares an entity named `smith` of the derived type `people`:

```
TYPE(people) :: smith = people(25,'John Smith')
```

F2008

Example 2

This example demonstrates the usage of the **TYPE()** type specifier to declare entities of intrinsic type.

```
TYPE(INTEGER) :: i
TYPE(INTEGER(KIND=2)) :: i2
TYPE(INTEGER(4)) :: i4

TYPE(CHARACTER(*)) :: cstar
TYPE(CHARACTER*2) :: c2
TYPE(CHARACTER(LEN=4,KIND=1)) :: c4
TYPE(CHARACTER(7)) :: c7

TYPE derived(l)
  TYPE(INTEGER), LEN :: l
  TYPE(CHARACTER*l) :: cl
  TYPE(COMPLEX), DIMENSION (l) :: cp
END TYPE derived
```

F2008

Related information

- Chapter 3, "Intrinsic data types," on page 37
- Chapter 4, "Derived types," on page 49
- "TYPE" on page 333
- "Constant expressions" on page 102
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28

Type Declaration

Purpose

A type declaration statement specifies the type, length, and attributes of objects and functions. You can assign initial values to objects.

F2003 A declaration type specification (*declaration_type_spec*) is used in a nonexecutable statement. **F2003**

Syntax

Type Declaration Statement (type_declaration_stmt)

```
►►──declaration_type_spec─────────────────────────────entity_decl_list────────►◄
                        └──::──┘
                        └─,──attr_spec_list──::─┘
```

where *declaration_type_spec* is as follows:

```
►►──┬──intrinsic_type_spec─────────────────────────────────────────────────────►◄
    │                                              (1)
    ├──TYPE──(──intrinsic_type_spec──)─────────────┤
    ├──TYPE──(──derived_type_spec──)───────────────┤
    │                    (2)
    ├──TYPE──(──*──)────────────────────────────────┤
    │                                         (3)
    ├──CLASS──(──derived_type_spec──)──────────────┤
    │                        (4)
    └──CLASS──(──*──)──────────────────────────────┘
```

Notes:

1 Fortran 2008

2 TS 29113

3 Fortran 2003

4 Fortran 2003
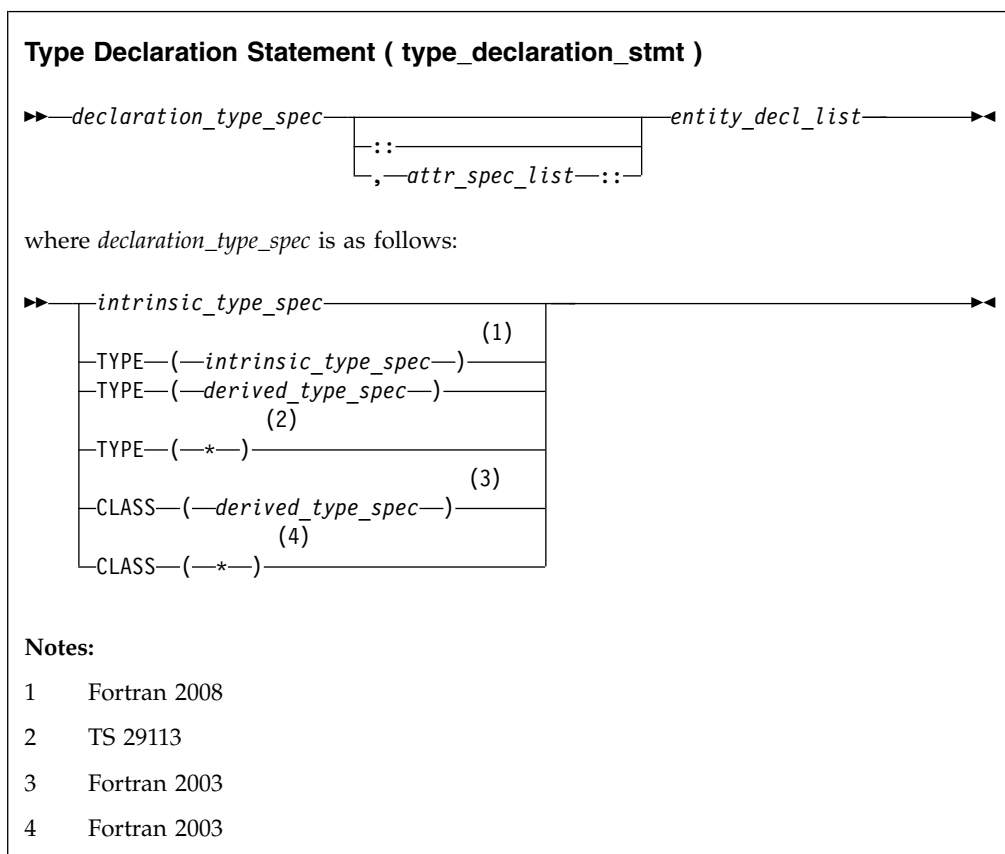
Parameters

- *intrinsic_type_spec* is any of the following types:

BYTE **1**	**INTEGER** [*kind_selector*]
CHARACTER [*char_selector*]	**LOGICAL** [*kind_selector*]
COMPLEX [*kind_selector*]	**REAL** [*kind_selector*]
DOUBLE COMPLEX	**VECTOR** (*vector_type_spec*) **1**
DOUBLE PRECISION	
1 IBM extension	

kind_selector

> represents one of the permissible length specifications for its associated type. **IBM** *int_literal_constant* cannot specify a kind type parameter. **IBM**
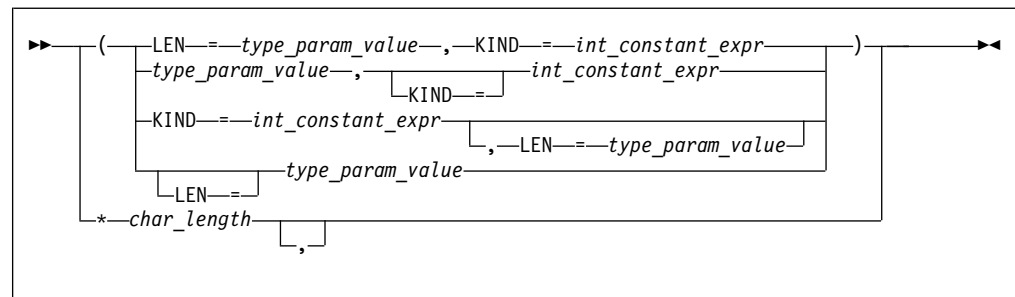
```
►►─┬─(─┬──────────────┬──int_constant_expr─)─┬──────────────────►◄
   │   └─KIND── = ─┘                         │
   │                              (1)        │
   └─ * ─int_literal_constant───────────────┘
```

Notes:

1   IBM extension.
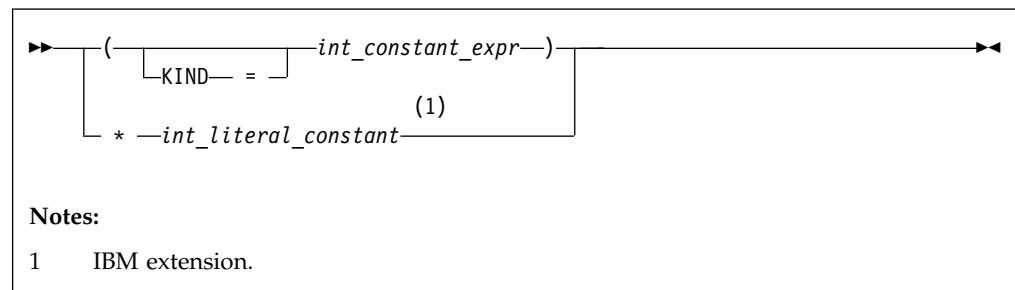
vector_type_spec

> must specify **PIXEL**, **REAL** of kind 4 or 8, or **INTEGER** or **UNSIGNED** of kind 1, 2, 4, or 8.

char_selector

> specifies the character length . **IBM** In XL Fortran, this is the number of characters between 0 and 256 MB. Values exceeding 256 MB are set to 256 MB, while negative values result in a length of zero. If not specified, the default length is 1. The kind type parameter, if specified, must be 1, which specifies the ASCII character representation. **IBM**

```
►►─┬─(─┬──LEN─=─type_param_value─,─KIND─=─int_constant_expr──────┬──)─┬──►◄
   │   │   ─type_param_value─,─┬──────────┬──int_constant_expr───┤    │
   │   │                       └─KIND─=─┘                        │    │
   │   │   ─KIND─=─int_constant_expr──┬────────────────────────┐ │    │
   │   │                              └─,─LEN─=─type_param_value─┘ │    │
   │   ┌─LEN─=─┐──type_param_value─────────────────────────────┘  │
   │   └─────────┘                                                 │
   └─*─char_length─┬────┬──────────────────────────────────────────┘
                   └─,─┘
```
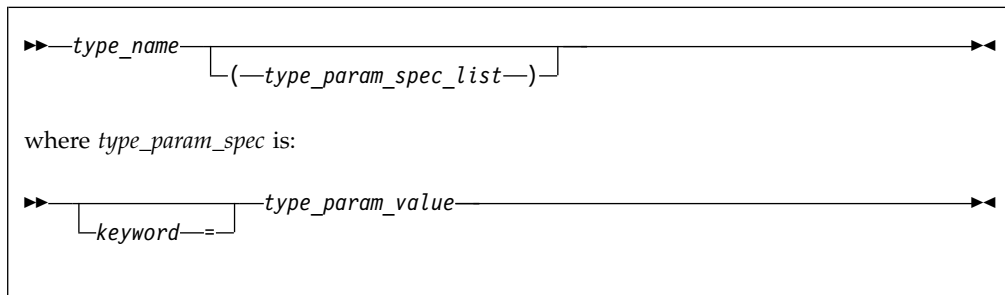
int_constant_expr

> is a scalar integer constant expression that must evaluate to 1

char_length

> is either a scalar integer literal constant (which cannot specify a kind type parameter) or a *type_param_value* enclosed in parentheses

- *derived_type_spec* is used to specify a particular derived type and type parameters.

```
►►─┬─type_name──────────────────────────┬─────────────────────────────►◄
                └─(─type_param_spec_list─)─┘

where type_param_spec is:

►►─┬──────────────┬─type_param_value─────────────────────────────────────►◄
     └─keyword──=─┘
```

where *type_param_spec* is:

type_param_spec
> is used to list type parameter values in the *derived_type_spec*.

keyword
> is a name of the parameter of the derived type listed in the derived type definition statement. Each parameter name cannot appear more than once in a *type_param_spec_list*. **F2003** When used with the **CLASS** keyword, the type must be extensible. If used with the **TYPE** keyword, the type must not be abstract. **F2003**

type_param_value
> is a colon (:), an asterisk (*), or an integer scalar expression. For a kind type parameter, the corresponding *type_param_value* must be an integer constant expression.
>
> Within a *derived_type_spec*, the *type_param_value* of an integer expression for a length type parameter must be a specification expression. A colon that specifies a deferred length type parameter, can only be used for a length parameter of an entity that has either a **POINTER** or **ALLOCATABLE** attribute. In this case, the value of a deferred type parameter is determined during program execution through either an **ALLOCATE** statement, an intrinsic assignment or a pointer assignment statement. An asterisk as a *type_param_value* specifies an assumed length type parameter. A *derived_type_spec* with an assumed length type parameter specifies a dummy argument, and the value of the assumed type parameter is that of the corresponding actual argument.
>
> **Note:** *type_param_value* is also used in *type_spec* that appears in **SELECT TYPE** constructs, **ALLOCATE** statements, or array constructors. In *type_spec*, a *type_param_value* that specifies a value for a length type parameter is not required to be a specification expression.

- *attr_spec* is any of the following attributes. For detailed rules about a particular attribute, refer to the statement of the same name.

ALLOCATABLE [1]	**PARAMETER**
ASYNCHRONOUS	**POINTER**
AUTOMATIC [2]	**PRIVATE**
BIND (C[, NAME=*binding_label*]) [1]	**PROTECTED** [1]
CONTIGUOUS [3]	**PUBLIC**
DIMENSION (*array_spec*)	**SAVE**
EXTERNAL	**STATIC** [2]

| INTENT (*intent_spec*)	TARGET
INTRINSIC	VALUE **1**
OPTIONAL	VOLATILE

Notes:
   1   Fortran 2003
   2   IBM extension
   3 Fortran 2008	

intent_spec
> is either **IN**, **OUT**, or **INOUT**

* is the asterisk indicator. An entity declared with `TS 29113` TYPE (*) `TS 29113` or CLASS(*) is unlimited polymorphic. An unlimited polymorphic entity is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity. `TS 29113` TYPE (*) is used to declare assumed-type objects to interoperate with C functions. `TS 29113`

:: is the double colon separator. Use the double colon separator when you specify attributes, =*constant_expr*, or => **NULL()**.

array_spec
> is a list of dimension bounds.

entity_decl

Notes:
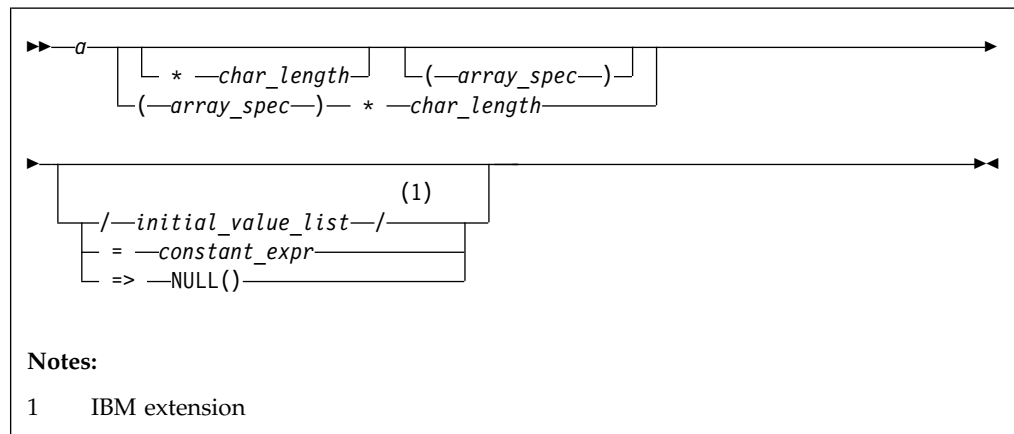
1   IBM extension

a is an object name or function name. *array_spec* cannot be specified for a function with an implicit interface.

char_length **(IBM extension)**
> overrides the length as specified in *kind_selector* and *char_selector*, and is only permitted in statements where the length can be specified with the initial keyword. A character entity can specify *char_length*, as defined above. A noncharacter entity can only specify an integer literal constant that represents one of the permissible length specifications for its associated type.

initial_value **(IBM extension)**
> provides an initial value for the entity specified by the immediately preceding name.

constant_expr
> provides a constant expression for the entity specified by the immediately preceding name.

=> **NULL()**
> provides the initial value for the pointer object.

Rules

Within the context of a derived type definition:
- If => appears in a component initialization, the **POINTER** attribute must appear in the *attr_spec_list*.
- If = appears in a component initialization, the **POINTER** attribute cannot appear in the component *attr_spec_list*.
- The compiler will evaluate *constant_expr* within the scoping unit of the type definition.

If => appears for a variable, the object must have the **POINTER** attribute.

If *constant_expr* appears for a variable, the object cannot have the **POINTER** attribute.

Entities in type declaration statements are constrained by the rules of any attributes specified for the entities, as detailed in the corresponding attribute statements.

The type declaration statement overrides the implicit type rules in effect. You can use a type declaration statement that confirms the type of an intrinsic function. The appearance of a generic or specific intrinsic function name in a type declaration statement does not cause the name to lose its intrinsic property.

An object cannot be initialized in a type declaration statement if it is a dummy argument, allocatable object, function result, object in a blank common block, integer pointer, external name, intrinsic name, or automatic object. Nor can an object be initialized if it has the **AUTOMATIC** attribute. The object may be initialized if it appears in a named common block in a block data program unit **IBM** or if it appears in a named common block in a module or submodule **IBM** .

In Fortran 95, a pointer can be initialized. Pointers can only be initialized by the use of **=> NULL()**.

The specification expression of a *type_param_value* or an *array_spec* can be a nonconstant expression if the specification expression appears in an interface body or in the specification part of a subprogram. Any object being declared that uses this nonconstant expression and is not a dummy argument or a pointee is called an *automatic object*.

An attribute cannot be repeated in a given type declaration statement, nor can an entity be explicitly given the same attribute more than once in a scoping unit.

constant_expr must be specified if the statement contains the **PARAMETER** attribute. If the entity you are declaring is a variable, and *constant_expr* or **NULL()** is specified, the variable is initially defined.

If the entity you are declaring is a derived type component, and *constant_expr* or **NULL()** is specified, the derived type has default initialization.

a becomes defined with the value determined by *constant_expr*, in accordance with the rules for intrinsic assignment. If the entity is an array, its shape must be specified either in the type declaration statement or in a previous specification statement in the same scoping unit. A variable or variable subobject cannot be initialized more than once. If *a* is a variable, the presence of *constant_expr* or **NULL()** implies that *a* is a saved object, except for an object in a named common block. The initialization of an object could affect the fundamental storage class of an object.

An *array_spec* specified in an *entity_decl* takes precedence over the *array_spec* in the **DIMENSION** attribute.

An array function result that does not have the ▸F2003▸ **ALLOCATABLE** or ◂F2003◂ **POINTER** attribute must have an explicit-shape array specification.

If the entity declared is a function, it must not have an accessible explicit interface unless it is an intrinsic function.

▸IBM▸ If T or F, defined previously as the name of a constant, appears in a type declaration statement, it is no longer an abbreviated logical constant but the name of the named constant. ◂IBM◂

The optional comma after *char_length* in a **CHARACTER** type declaration statement is permitted only if no double colon separator (::) appears in the statement.

▸F2003▸ If the **CHARACTER** type declaration statement specifies a length of a colon, the length type parameter is a *deferred type parameter*. An entity or component with a deferred type parameter must specify the **ALLOCATABLE** or **POINTER** attribute. A deferred type parameter is a length type parameter whose value can change during the execution of the program. ◂F2003◂

If the **CHARACTER** type declaration statement is in the scope of a module, ▸F2008▸ submodule ◂F2008◂ , block data program unit, or main program, and you specify the length of the entity as an inherited length, the entity must be the name of a named character constant. The character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute.

If the **CHARACTER** type declaration statement is in the scope of a procedure and the length of the entity is inherited, the entity name must be the name of a dummy argument or a named character constant. If the statement is in the scope of an external function, it can also be the function or entry name in a **FUNCTION** or **ENTRY** statement in the same program unit. If the entity name is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the procedure. If the entity name is the name of a character constant, the character constant assumes the length of its corresponding expression defined by the **PARAMETER** attribute. If the entity name is a function or entry name, the entity assumes the length specified in the calling scoping unit.

The length of a character function can be a specification expression (which must be a constant expression if the function type is not declared in an interface block) or it is a colon, or an asterisk, indicating the length of a dummy procedure name. The length cannot be an asterisk if the function is an internal or module function, if it is recursive, or if it returns array or pointer values.

Examples

```
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER*7, TARGET :: ORANGES = 'ORANGES'
CALL TEST(APPLES)

SUBROUTINE TEST(VARBL)
  CHARACTER*(*), OPTIONAL :: VARBL   ! VARBL inherits a length of 6

  COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
  REAL, POINTER :: XCONST

  TYPE PEOPLE                        ! Defining derived type PEOPLE
    INTEGER AGE
    CHARACTER*20 NAME
  END TYPE PEOPLE
  TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END SUBROUTINE
```

The following example illustrates the use of derived types parameters in a declaration with a derived type. See the examples in "TYPE" on page 333 for the type definitions themselves.

```
! Use of the types declared in the example in section Derived type.
TYPE(MULTIDIM(8,3)) :: LOCATION = MULTIDIM(8,3)([1.1_8,2.2_8,3.3_8])
TYPE(NAMED_MULTI(8,3,12)) :: MY_SPOT
    = NAMED_MULTI(8,3,12)([REAL(8):: 1.1,2.2,3.3],"You are here")


! "PEOPLE" can be defined using type parameters:
TYPE PEOPLE (AGE_KIND, NAME_LEN)
  INTEGER, KIND  :: AGE_KIND
  INTEGER, LEN   :: NAME_LEN
  INTEGER(AGE_KIND) :: AGE
  CHARACTER(NAME_LEN) :: NAME
END TYPE PEOPLE

! Use integer(2) for age, character(20) for name:
TYPE (PEOPLE(2,20)) :: SMITH = PEOPLE(2,20)(25,'John Smith')

! Use integer(1) for age, deferred length for name:
TYPE (PEOPLE(1,:)), ALLOCATABLE :: JDOE
! Actually allocate JDOE with a name of length 8 using implicit allocation:
JDOE = PEOPLE(1,8)(22, "John Doe")
! Explicitly deallocate and reallocate JDOE with a different length:
DEALLOCATE(JDOE)
ALLOCATE(PEOPLE(1,15) :: JDOE)
```

The following example illustrates the declaration of a vector.

```
VECTOR (REAL(4)) :: vector_object
```

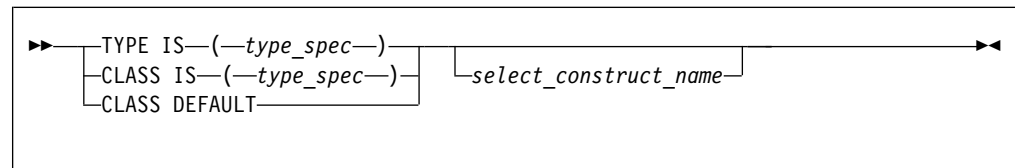## Related information

- "Constant expressions" on page 102
- TYPE
- "Determining Type" on page 18, for details on the implicit typing rules
- "Array declarators" on page 76
- "Automatic objects" on page 20
- "Storage classes for variables (IBM extension)" on page 28
- "DATA" on page 327, for details on initial values
- "Polymorphic entities (Fortran 2003)" on page 20
- "CLASS (Fortran 2003)" on page 312

Type Guard (Fortran 2003)

Purpose

A type guard statement initiates a type guard statement block in a **SELECT TYPE** construct. A **SELECT TYPE** construct can have any number of statement blocks, only one of which is selected for execution. The selection is based on the dynamic type and the kind type parameters of an expression — the *selector* — in a **SELECT TYPE** statement, the type and the corresponding kind type parameters of each type guard statement.

Syntax

```
►►─┬─TYPE IS─(─type_spec─)──────┬──────────────────────────────────►◄
   ├─CLASS IS─(─type_spec─)─────┤  ┌─select_construct_name─┐
   └─CLASS DEFAULT──────────────┘
```

type_spec
> must be an extensible derived type or intrinsic type. The length type parameters must be assumed.

select_construct_name
> is a name that identifies the **SELECT TYPE** construct

Rules

If the selector of the **SELECT TYPE** statement is not unlimited polymorphic, the *type_spec* must specify an extension of the declared type of the selector.

For a single **SELECT TYPE** construct, the same type and kind type parameter values must not be specified in more than one **TYPE IS** type guard statement and must not be specified in more than one **CLASS IS** type guard statement.

The **CLASS DEFAULT** type guard statement can only occur once in a **SELECT TYPE** construct.

If the *select_construct_name* is specified, it must match the name specified on the **SELECT TYPE** and **END SELECT** statements.
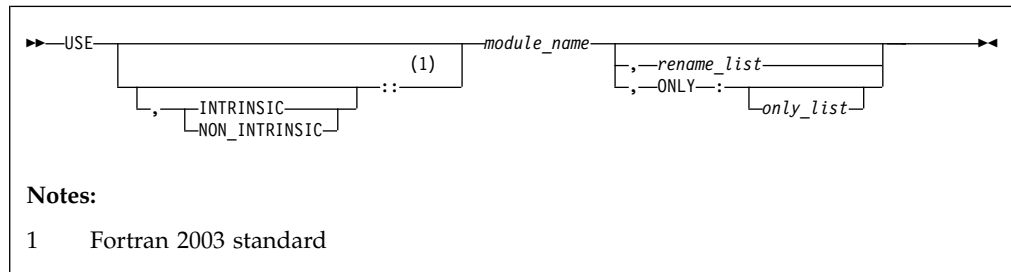
Related information
- "SELECT TYPE construct (Fortran 2003)" on page 147
- "SELECT TYPE (Fortran 2003)" on page 460
- "END (Construct)" on page 351, for details on the **END SELECT** statement

USE

Purpose

The **USE** statement is a module reference that provides local access to the public entities of a module.

Syntax

```
>>─USE─┬──────────────────────────┬─ module_name ─┬──────────────────┬──><
       │                    (1)    │               ├─,─ rename_list ──┤
       └─,─┬─INTRINSIC─────┬─ :: ──┘               └─,─ONLY─:─┬───────┤
           └─NON_INTRINSIC─┘                                  └─only_list─┘
```

Notes:

1   Fortran 2003 standard

rename is

- the assignment of a local name to an accessible data entity: `local-name => use-name`

- **F2003** renaming a use-defined operator to a local-defined operator: `OPERATOR(local-defined-operator) => OPERATOR(use-defined-operator)`

  F2003

only   is a *rename*, a generic specification, or the name of a variable, procedure, derived type, named constant, or namelist group

Rules

The **USE** statement can only appear prior to all other statements in *specification_part*. Multiple **USE** statements may appear within a scoping unit.

IBM At the time the file containing the **USE** statement is being compiled, the specified module must precede the **USE** statement in the file or the module must have been already compiled in another file. Each referenced entity must be the name of a public entity in the module. **IBM**

Entities in the scoping unit become *use-associated* with the module entities, and the local entities have the attributes of the corresponding module entities.

Fortran 2003

By default, either an intrinsic module or a non-intrinsic module with the specified name is accessed. If both an intrinsic module and a non-intrinsic module have this name, the non-intrinsic module is accessed. If you specify **INTRINSIC** or **NON_INTRINSIC**, only an intrinsic module or only a non-intrinsic module can be accessed.

When you rename an operator in a *rename-list* or an *only-list*, the *use-defined-operator* is identified by the *local-defined-operator* for the scoping unit that contains the **USE** statement. That operator must be a public entity that is not a generic binding within the module you specify in the **USE** statement.

End of Fortran 2003

In addition to the **PRIVATE** attribute, the **ONLY** clause of the **USE** statement provides further constraint on which module entities can be accessed. If the **ONLY** clause is specified, only entities named in the *only_list* are accessible. If no list follows the keyword, no module entities are accessible. If the **ONLY** clause is absent, all public entities are accessible.

If a scoping unit contains multiple **USE** statements, all specifying the same module, and one of the statements does not include the **ONLY** clause, all public entities are accessible. If each **USE** statement includes the **ONLY** clause, only those entities named in one or more of the *only_lists* are accessible.

You can rename an accessible entity for local use. A module entity can be accessed by more than one local name. If no renaming is specified, the name of the use-associated entity becomes the local name. The local name of a use-associated entity cannot be redeclared. However, if the **USE** statement appears in the scoping unit of a module, the local name can appear in a **PUBLIC** or **PRIVATE** statement.

If multiple generic interfaces that are accessible to a scoping unit have the same local name, operator, or assignment, they are treated as a single generic interface. In such a case, one of the generic interfaces can contain an interface body to an accessible procedure with the same name. Otherwise, any two different use-associated entities can only have the same name if the name is not used to refer to an entity in the scoping unit. If a use-associated entity and host entity share the same name, the host entity becomes inaccessible through host association by that name.

The accessed entities have the attributes specified in the module, except that an entity may have a different accessibility attribute or it can have the **VOLATILE** attribute in the local scoping unit even if the associated module entity does not.

A module must not reference itself, either directly or indirectly. For example, module X cannot reference module Y if module Y references module X.

Consider the situation where a module (for example, module B) has access through use association to the public entities of another module (for example, module A). The accessibility of module B's local entities (which includes those entities that are use-associated with entities from module A) to other program units is determined by the **PRIVATE** and **PUBLIC** attributes, or, if absent, through the default accessibility of module B. Of course, other program units can access the public entities of module A directly.

Examples

```
MODULE A
  REAL :: X=5.0
END MODULE A
MODULE B
  USE A
  PRIVATE :: X                ! X cannot be accessed through module B
  REAL :: C=80, D=50
END MODULE B
PROGRAM TEST
  INTEGER :: TX=7
  CALL SUB
  CONTAINS

  SUBROUTINE SUB
  USE B, ONLY : C
  USE B, T1 => C
  USE B, TX => C              ! C is given another local name
  USE A
  PRINT *, TX                 ! Value written is 80 because use-associated
                              ! entity overrides host entity
  END SUBROUTINE
END
```

Example: Renaming a defined operator **(Fortran 2003)**

```
module temp
 type real_num
 real :: x
 end type

 interface operator (.add.)
 module procedure real_add
 end interface

 contains
 funtion real_add(a,b)
 type(real_num) :: real_add
 type(real_num), intent(in) :: a,b
 real_add%x = a%x+b%x
 end function real_add

end module

program main
use temp , operator(.plus.) => operator(.add.)
type(real_num) :: a,b,c
c=a.plus.b
end program
```

Example: Invalid because operator has a private attribute

```
module temp
 type real_num
 real :: x
 end type

 interface operator (.add.)
 module procedure real_add
 end interface

 private :: operator(.add.) !operator is given the private attribute

 contains
 function real_add(a,b)
 type(real_num) :: real_add
 type(real_num), intent(in) :: a,b
 real_add%x = a%x+b%x
 end function real_add

 contains

 end module

program main
!operator cannot be renamed because it has a private attribute.
use temp , operator(.plus.) => operator(.add.)
type(real_num) :: a,b,c
c=a.plus.b
end program
```

The following example is invalid:

```
Module mod1
    use, intrinsic :: ieee_exceptions
end Module

Module mod2
   use, non_intrinsic :: ieee_exceptions
end Module

Program invalid_example
```

```
      use mod1
      use mod2
! ERROR: a scoping unit must not access an
! intrinsic module and a non-intrinsic module
! with the same name.

end program
```

Related information
- "Modules" on page 177
- "PRIVATE" on page 432
- "VOLATILE" on page 489
- "PUBLIC" on page 440
- "Order of statements and execution sequence" on page 16

VALUE (Fortran 2003)

Purpose

The **VALUE** attribute specifies an argument association between a dummy and an actual argument. This association allows you to pass the dummy argument with the value of the actual argument. This Fortran 2003 pass by value implementation provides a standard conforming option to the **%VAL** built-in function.

An actual argument and the associated dummy argument can change independently. Changes to the value or definition status of the dummy argument do not affect the actual argument. A dummy argument with the **VALUE** attribute becomes associated with a temporary variable with an initial value identical to the value of the actual argument.

Syntax

```
►►──VALUE──────────dummy_argument_name_list──────────────────────►◄
              └─::─┘
```
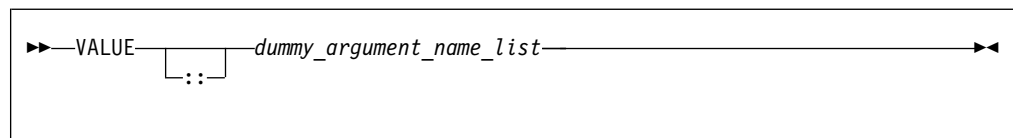
Rules

You must specify the **VALUE** attribute for dummy arguments only.

You must not use the **%VAL** or **%REF** built-in functions to reference a dummy argument with the **VALUE** attribute, or the associated actual argument.

A referenced procedure that has a dummy argument with the **VALUE** attribute must have an explicit interface.

A dummy argument with the **VALUE** attribute can be of character type .

You must not specify the **VALUE** attribute for the following items:
- Dummy procedures
- Polymorphic items
- TS 29113 Assumed-rank objects TS 29113

F2008 You can specify the **VALUE** attribute on an array dummy argument that has either assumed-shape or explicit-shape. However, you cannot specify the **VALUE** attribute on an array dummy argument that has deferred-shape, assumed-size, or implied-shape. F2008

Table 55. Attributes compatible with the VALUE attribute

INTENT(IN)	OPTIONAL	TARGET

If a dummy argument has both the **VALUE** and **TARGET** attributes, any pointers associated with that dummy argument become undefined after the execution of the procedure.

Examples

```
Program validexm1
  integer :: x = 10, y = 20
  print *, 'before calling: ', x, y
  call intersub(x, y)
  print *, 'after calling: ', x, y

  contains
  subroutine intersub(x,y)
    integer, value ::  x
    integer y
    x = x + y
    y = x*y
    print *, 'in subroutine after changing: ', x, y
  end subroutine
end program validexm1
```

Expected output:

```
before calling: 10 20
in subroutine after changing: 30 600
after calling: 10 600
```

Related information

"Assumed-shape arrays" on page 79

"Explicit-shape arrays" on page 77

%VAL

VECTOR (IBM extension)

Purpose

A **VECTOR** type declaration statement specifies that one or more entities have a vector type.

Syntax

You can declare a vector using **VECTOR**(*type_spec*) as part of a type declaration statement. The type declaration statement contains the complete syntax for declaring a vector data type. In a **VECTOR**(*type_spec*), *type_spec* must specify **PIXEL**, **REAL** of kind 4 or 8, or **INTEGER** or **UNSIGNED** of kind 1, 2, 4, or 8.

VIRTUAL (IBM extension)

Purpose

The **VIRTUAL** statement specifies the name and dimensions of an array. It is an alternative form of the **DIMENSION** statement, although there is no **VIRTUAL** attribute.

Syntax

```
►►──VIRTUAL──array_declarator_list──────────────────────────────►◄
```

array_declarator_list
>   is a list of array declarators that specifies the rank or rank and shape of an array.

Rules

You can specify arrays with a maximum of 20 dimensions

Only one array specification for an array name can appear in a scoping unit.

Examples

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

Related information
* Chapter 5, "Array concepts," on page 75
* "DIMENSION" on page 335

VOLATILE

Purpose

The **VOLATILE** attribute is used to designate a data object as being mapped to memory that can be accessed by independent input/output processes and independent, asynchronously interrupting processes. Code that manipulates volatile data objects is not optimized.

Syntax

```
                           ┌────────,───────┐
                           │                │
►►──VOLATILE──┬─────┬──────▼──┬─variable_name───────────┬──►◄
              └─::──┘         ├─/─common_block_name─/────┤
                             └─derived_type_name────────┘
```

Rules

If an array name is declared volatile, each element of the array is considered volatile. If a common block is declared volatile, each variable in the common block is considered volatile. An element of a common block can be declared volatile without affecting the status of the other elements in the common block.

If a common block is declared in multiple scopes, and if it (or one or more of its elements) is declared volatile in one of those scopes, you must specify the **VOLATILE** attribute in each scope where you require the common block (or one or more of its elements) to be considered volatile.

If a derived type name is declared volatile, all variables declared with that type are considered volatile. If an object of derived type is declared volatile, all of its components are considered volatile. If a component of a derived type is itself derived, the component does not inherit the volatile attribute from its type. A derived type name that is declared volatile must have had the **VOLATILE** attribute prior to any use of the type name in a type declaration statement.

If a pointer is declared volatile, the storage of the pointer itself is considered volatile. The **VOLATILE** attribute has no effect on any associated pointer targets.

If you declare an object to be volatile and then use it in an **EQUIVALENCE** statement, all of the objects that are associated with the volatile object through equivalence association are considered volatile.

Any data object that is shared across threads and is stored and read by multiple threads must be declared as **VOLATILE**. If, however, your program only uses the automatic or directive-based parallelization facilities of the compiler, variables that have the **SHARED** attribute need not be declared **VOLATILE**.

If the actual argument associated with a dummy argument is a variable that is declared volatile, you must declare the dummy argument volatile if you require the dummy argument to be considered volatile. If a dummy argument is declared volatile, and you require the associated actual argument to be considered volatile, you must declare the actual argument as volatile.

Declaring a statement function as volatile has no effect on the statement function.

Within a function subprogram, the function result variable can be declared volatile. Any entry result variables will be considered volatile. An **ENTRY** name must not be specified with the **VOLATILE** attribute.

`F2008` An object can have the **VOLATILE** attribute inside a **BLOCK** construct, regardless of whether the object has the **VOLATILE** attribute outside the **BLOCK** construct. `F2008`

Fortran 2003

Using **-qxlf2003=volatile**

If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has the **VOLATILE** attribute, that dummy argument shall be an assumed-shape array.

If an actual argument is a pointer array, and the corresponding dummy argument has the **VOLATILE** attribute, that dummy argument shall be an assumed-shape array or a pointer array.

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have the **VOLATILE** attribute.

Host associated entities are known by the same name and have the same attributes as in the host, except that an accessed entity may have the **VOLATILE** attribute even if the host entity does not.

In an internal or module procedure, if a variable that is accessible via host association is specified in a **VOLATILE** statement, that host variable is given the **VOLATILE** attribute in the local scope.

A use associated entity may have the **VOLATILE** attribute in the local scoping unit even if the associated module entity does not.

───────────── **End of Fortran 2003** ─────────────

Table 56. Attributes compatible with the VOLATILE attribute

ALLOCATABLE **1**	INTENT	PUBLIC
ASYNCHRONOUS	OPTIONAL	SAVE
AUTOMATIC **3**	POINTER	STATIC **3**
CONTIGUOUS **2**	PRIVATE	TARGET
DIMENSION	PROTECTED **1**	
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension		

Examples

```
      FUNCTION TEST ()
        REAL ONE, TWO, THREE
        COMMON /BLOCK1/A, B, C
        ...
        VOLATILE /BLOCK1/, ONE, TEST
! Common block elements A, B and C are considered volatile
! since common block BLOCK1 is declared volatile.
        ...
        EQUIVALENCE (ONE, TWO), (TWO, THREE)
! Variables TWO and THREE are volatile as they are equivalenced
! with variable ONE which is declared volatile.
      END FUNCTION
```

Related information
- "Direct access" on page 217

WAIT (Fortran 2003)

Purpose

The **WAIT** statement may be used to wait for an asynchronous data transfer to complete or it may be used to detect the completion status of an asynchronous data transfer statement.

Syntax

```
►►──WAIT──(──wait_list──)──────────────────────────────────────►◄
```

wait_list
> is a list that must contain one **ID=** specifier and at most one of each of the other valid specifiers. The valid specifiers are:

ASYNCHRONOUS=*char_expr* **(Fortran 2003)**
> allows execution to continue without waiting for the data transfer to complete. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**.
>
> If a **DONE=** specifier appears, an **ID=** specifier must also appear. If the **ID=** specifier appears, a wait operation for the specified data transfer operation is performed. If there is no **ID=** specifier then wait operations for all pending data transfers for the specified unit are performed. Execution of a file positioning statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

DONE= *logical_variable*
> specifies whether or not the asynchronous I/O statement is complete. If the **DONE=** specifier is present, the *logical_variable* is set to true if the asynchronous I/O is complete and is set to false if it is not complete. If the returned value is false, then one or more **WAIT** statements must be executed until either the **DONE=** specifier is not present, or its returned value is true. A **WAIT** statement without the **DONE=** specifier, or a **WAIT** statement that sets the *logical_variable* value to true, is the matching **WAIT** statement to the data transfer statement identified by the same **ID=** value.

END= *stmt_label*
> is an end-of-file specifier that specifies a statement label at which the program is to continue if an endfile record is encountered and no error occurs. If an external file is positioned after the endfile record, the **IOSTAT=** specifier, if present, is assigned a negative value, and the **NUM=** specifier, if present, is assigned an integer value. Coding the **END=** specifier suppresses the error message for end-of-file. This specifier can be specified for a unit connected for either sequential or direct access.
>
> The *stmt_label* defined for the **END=** specifier of the asynchronous data transfer statement need not be identical to the *stmt_label* defined for the **END=** specifier of the matching **WAIT** statement.

ERR= *stmt_label*
> is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in case of an error. Coding the **ERR=** specifier suppresses error messages.
>
> The *stmt_label* defined for the **ERR=** specifier of the asynchronous data transfer statement need not be identical to the *stmt_label* defined for the **ERR=** specifier of the matching **WAIT** statement.

ID= *integer_expr*
> indicates the data transfer with which this **WAIT** statement is identified. The *integer_expr* is an integer expression of type **INTEGER(4)** or default integer. To initiate an asynchronous data transfer, the **ID=** specifier is used on a **READ** or **WRITE** statement.

IOMSG= *iomsg_variable*

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

is an input/output status specifier that specifies the status of the input/output operation. *ios* is an integer variable. When the input/output statement containing this specifier finishes execution, *ios* is defined with:

- A zero value if no error condition occurs.
- A positive value if an error occurs.
- A negative value if an end-of-file condition is encountered and no error occurs.

The *ios* defined for the **IOSTAT=** specifier of the asynchronous data transfer statement is not required to be identical to the *ios* defined for the **IOSTAT=** specifier of the matching **WAIT** statement.

Rules

The matching **WAIT** statement must be in the same scoping unit as the corresponding asynchronous data transfer statement. Within the instance of that scoping unit, the program must not execute a **RETURN**, **END**, or **STOP** statement before the matching **WAIT** statement is executed.
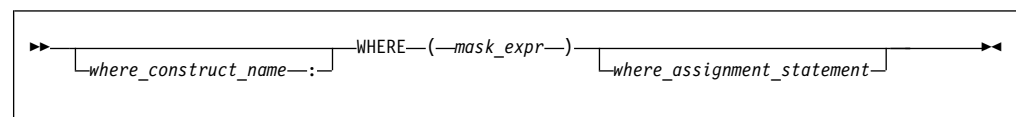
Related information

- "Asynchronous Input/Output" on page 220
- *Implementation details of XL Fortran Input/Output* in the *XL Fortran Optimization and Programming Guide*

WHERE

Purpose

The **WHERE** statement masks the evaluation of expressions and assignments of values in array assignment statements. It does this according to the value of a logical array expression. The **WHERE** statement can be the initial statement of the **WHERE** construct.

Syntax

```
►►─┬────────────────────────┬─ WHERE ─(─ mask_expr ─)─┬──────────────────────────┬─►◄
   └─ where_construct_name ─:─┘                        └─ where_assignment_statement ─┘
```

mask_expr

is a logical array expression

where_construct_name
   is a name that identifies the **W HERE** construct

Rules

If a *where_assignment_statement* is present, the **WHERE** statement is not the first statement of a **WHERE** construct. If a *where_assignment_statement* is absent, the **WHERE** statement is the first statement of the **WHERE** construct, and is referred to as a **WHERE** construct statement. An **END WHERE** statement must follow. See "WHERE construct" on page 121 for more information.

If the **WHERE** statement is not the first statement of a **WHERE** construct, you can use it as the terminal statement of a **DO** or **DO WHILE** construct.

You can nest **WHERE** statements within a **WHERE** construct. A *where_assignment_statement* that is a defined assignment must be an elemental defined assignment.

In each *where_assignment_statement*, the *mask_expr* and the *variable* being defined must be arrays of the same shape. Each *mask_expr* in a **WHERE** construct must have the same shape.

A **WHERE** statement that is part of a *where_body_construct* must not be a branch target statement.

The execution of a function reference in the *mask_expr* of a **WHERE** statement can affect entities in the *where_assignment_statement*.

See "Interpreting masked array assignments" on page 123 for information on interpreting mask expressions.

If a *where_construct_name* appears on a **WHERE** construct statement, it must also appear on the corresponding **END WHERE** statement. A construct name is optional on any masked **ELSEWHERE** and **ELSEWHERE** statements in the **WHERE** construct.

A *where_construct_name* can only appear on a **WHERE** construct statement.

Examples

```
REAL, DIMENSION(10) :: A,B,C

!   In the following WHERE statement, the LOG of an element of A
!   is assigned to the corresponding element of B only if that
!   element of A is a positive value.

WHERE (A>0.0) B = LOG(A)

:
END
```

The following example shows an elemental defined assignment in a **WHERE** statement:

```
INTERFACE ASSIGNMENT(=)
  ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
    LOGICAL, INTENT(OUT) :: X
    REAL, INTENT(IN) :: Y
  END SUBROUTINE MY_ASSIGNMENT
END INTERFACE
```

```
INTEGER A(10)
REAL C(10)
LOGICAL L_ARR(10)

C = (/ -10., 15.2, 25.5, -37.8, 274.8, 1.1, -37.8, -36.2, 140.1, 127.4 /)
A = (/ 1, 2, 7, 8, 3, 4, 9, 10, 5, 6 /)
L_ARR = .FALSE.

WHERE (A < 5) L_ARR = C

! DATA IN ARRAY L_ARR AT THIS POINT:
!
! L_ARR = F, T, F, F, T, T, F, F, F, F

END

ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
  LOGICAL, INTENT(OUT) :: X
  REAL, INTENT(IN) :: Y

  IF (Y < 0.0) THEN
    X = .FALSE.
  ELSE
    X = .TRUE.
  ENDIF
END SUBROUTINE MY_ASSIGNMENT
```
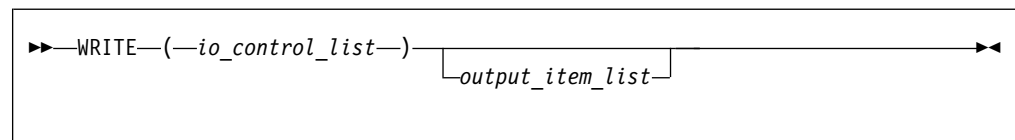
Related information

- "WHERE construct" on page 121
- "ELSEWHERE" on page 348
- "END (Construct)" on page 351, for details on the **END WHERE** statement

---

WRITE

Purpose

The **WRITE** statement is a data transfer output statement.

Syntax

```
►►──WRITE──(──io_control_list──)──┬─────────────────┬──►◄
                                  └─output_item_list─┘
```

output_item

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

- A variable name. An array is treated as if all of its elements were specified in the order in which they are arranged in storage.

  A pointer must be associated with a target, and an allocatable object must be allocated. A derived-type object cannot have any ultimate component that is outside the scoping unit of this statement. The evaluation of *output_item* cannot result in a derived-type object that contains a pointer. The structure components of a structure in a formatted statement are treated as if they appear in the order of the

derived-type definition; in an unformatted statement, the structure components are treated as a single value in their internal representation (including padding).

- An expression
- An implied-**DO** list, as described under "Implied-DO List" on page 501

  F2003 An *output_item* must not be a procedure pointer. **F2003**

io_control

is a list that must contain one unit specifier (**UNIT=**), and can also contain one of each of the other valid specifiers:

[UNIT=] *u*

is a unit specifier that specifies the unit to be used in the 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 expression whose value is in the range 1 through 2147483647
- **IBM** An asterisk, which identifies external unit 6 and is preconnected to standard output **IBM**
- **F2008** A NEWUNIT value **F2008**

An internal file identifier refers to an internal file. It is the name of a character variable, which cannot be an array section with a vector subscript.

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

[FMT=] *format*

is a format specifier that specifies the format to be used in the output operation. *format* is a format identifier that can be:

- The statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.
- The name of a scalar **INTEGER(4)** or **INTEGER(8)** variable that was assigned the statement label of a **FORMAT** statement. The **FORMAT** statement must be in the same scoping unit.

  Fortran 95 does not permit assigning of a statement label.

- A character constant enclosed in parentheses. Only the format codes listed under "FORMAT" on page 376 can be used between the parentheses. Blank characters can precede the left parenthesis or follow the right parenthesis.
- A character variable 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. If *format* is an array element, the format identifier must not exceed the length of the array element.
- An array of noncharacter intrinsic type. The data must be a valid format identifier as described under character array.
- Any character expression, except one involving concatenation of an operand that specifies inherited length, unless the operand is the name of a constant.
- An asterisk, specifying list-directed formatting.

- A namelist specifier that specifies the name of a namelist list that you have previously defined.

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 with **UNIT=** omitted. **NML=** and **FMT=** cannot both be specified in the same output statement.

ASYNCH= *char_expr* **(IBM extension)**
is an asynchronous I/O specifier that indicates whether an explicitly connected unit is to be used for asynchronous I/O.

char_expr is a scalar character expression whose value is either **YES** or **NO**. **YES** specifies that asynchronous data transfer statements are permitted for this connection. **NO** specifies that asynchronous data transfer statements are not permitted for this connection. The value specified will be in the set of transfer methods permitted for the file. If this specifier is omitted, the default value is **NO**.

Preconnected units are connected with an **ASYNCH=** value of **NO**.

The **ASYNCH=** value of an implicitly connected unit is determined by the first data transfer statement performed on the unit. If the first statement performs an asynchronous data transfer and the file being implicitly connected permits asynchronous data transfers, the **ASYNCH=** value is **YES**. Otherwise, the **ASYNCH=** value is **NO**.

ASYNCHRONOUS= *char_expr* **(Fortran 2003)**
allows execution to continue without waiting for the data transfer to complete. *char_expr* is a scalar character expression that must evaluate to **YES** or **NO**. **ASYNCHRONOUS=YES** must not appear unless **UNIT=** specifies a file unit number. If **ID=** appears, an **ASYNCHRONOUS=YES** must also appear.

A statement and the I/O operation are synchronous if **ASYNCHRONOUS=NO** or if both **ASYNCHRONOUS=** and **ID=** are absent. For **ASYNCHRONOUS=YES** or if **ID=** appears, asynchronous I/O is permitted only for external files opened with **ASYNCHRONOUS=YES** in the **OPEN** statement.

If a variable is used in an asynchronous data transfer statement as an item in an I/O list, a group object in a namelist or as a **SIZE=** specifier, the base object of the *data_ref* is implicitly given the **ASYNCHRONOUS** attribute in the scoping unit of the data transfer statement. For asynchronous nonadvancing input, the storage units specified in the **SIZE=** specifier become defined with the count of the characters transferred when the corresponding wait operation is executed. For asynchronous output, a pending I/O storage sequence affector shall not be redefined, become undefined, or have its pointer association status changed. For asynchronous input, a pending I/O storage sequence affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the **VALUE** attribute, or have its pointer association status changed.

When an error, end-of-file or end-of-record condition occurs for a previously executed asynchronous data transfer statement , a wait operation is performed for all pending data transfer operations on that unit. When a condition occurs during a subsequent statement, any actions specified by **IOSTAT=**, **IOMSG=**, **ERR=**, **END=**, and **EOR=** specifiers for that statement are taken.

A wait operation is performed by a **WAIT**, **CLOSE**, or file positioning statement.

DECIMAL= *char_expr***(Fortran 2003)**
> temporarily changes the default *decimal edit mode* for the duration of an I/O statement. *char_expr* is a scalar character expression whose value must evaluate to either **POINT**, or **COMMA**. After each **WRITE** statement, the mode defaults to whatever decimal mode was specified (or assumed) on the **OPEN** statement for that unit.

POS=*integer_expr***(Fortran 2003)**
> *integer_expr* is an integer expression greater than 0. **POS=** specifies the file position of the file storage unit to be written in a file connected for stream access. You must not use **POS=** for a file that cannot be positioned.

REC= *integer_expr*
> is a record specifier that specifies the number of the record to be written in a file connected for direct access. The **REC=** specifier is only permitted for direct output. *integer_expr* is an integer expression whose value is positive. A record specifier is not valid if formatting is list-directed or if the unit specifier specifies an internal file. The record specifier represents the relative position of a record within a file. The relative position number of the first record is 1. You must not specify **REC=** in data transfer statements that specify a unit connected for stream access, or use the **POS=** specifier.

ROUND= *char-expr***(Fortran 2003)**

> temporarily changes the current value of the I/O rounding mode for the duration of this I/O statement. If omitted, then the rounding mode is unchanged. *char-expr* evaluates to either **UP**, **DOWN**, **ZERO**, **NEAREST**, **COMPATIBLE** or **PROCESSOR_DEFINED**

> The rounding mode helps specify how decimal numbers are converted to an internal representation, (that is, in binary) from a character representation and vice versa during formatted input and output. The rounding modes have the following functions:
> * In the **UP** rounding mode the value from the conversion is the smallest value that is greater than or equal to the original value.
> * In the **DOWN** rounding mode the value from the conversion is the greatest value that is smaller than or equal to the original value.
> * In the **ZERO** rounding mode the value from the conversion is the closest value to the original value, and not greater in magnitude.
> * In the **NEAREST** rounding mode the value from the conversion is the closer of the two nearest representable values. If both values are equally close then the even value will be chosen. In IEEE rounding conversions, NEAREST corresponds to the ieee_nearest rounding mode as specified by the IEEE standard.
> * In the **COMPATIBLE** rounding mode the value from the conversion is the closest of the two nearest representable values, or the value further away from zero if halfway between.
> * In the **PROCESSOR_DEFINED** rounding mode the value from the conversion is processor dependent and may correspond to the other modes. In the **PROCESSOR_DEFINED** rounding mode the value from the conversion is processor dependent and may correspond to the other modes. In XL Fortran, the **PROCESSOR_DEFINED** rounding mode will

be the rounding mode you choose in the floating-point control register. If you do not set the floating-point control register explicitly, the default rounding mode is **NEAREST**.

SIGN= *char_expr***(Fortran 2003)**

indicates the sign mode in effect for a connection for formatted input/output. If *char_expr* is assigned the value **PLUS**, the processor shall produce a plus sign in any position that normally contains an optional plus sign and suppresses plus signs in these positions if *char_expr* is assigned the value **SUPPRESS**. *char_expr* can also be assigned the value **PROCESSOR_DEFINED** which is the default sign mode and acts the same as **SUPPRESS**. If there is no connection, or if the connection is not for formatted input/output, *char_expr* is assigned the value **UNDEFINED**.

IOMSG= *iomsg_variable***(Fortran 2003)**

is an input/output status specifier that specifies the message returned by the input/output operation. *iomsg_variable* is a scalar default character variable. It must not be a use-associated nonpointer protected variable. When the input/output statement containing this specifier finishes execution, *iomsg_variable* is defined as follows:

- If an error, end-of-file, or end-of-record condition occurs, the variable is assigned an explanatory message as if by assignment.
- If no such condition occurs, the value of the variable is unchanged.

IOSTAT= *ios*

is an input/output status specifier that specifies the status of the input/output operation. *ios* is an integer variable. Coding the **IOSTAT=** specifier suppresses error messages. When the statement finishes execution, *ios* is defined with:

- A zero value if no error condition occurs
- A positive value if an error occurs.

ID= *integer_variable***(IBM extension)**

indicates that the data transfer is to be done asynchronously. The *integer_variable* is an integer variable. If no error is encountered, the *integer_variable* is defined with a value after executing the asynchronous data transfer statement. This value must be used in the matching **WAIT** statement.

Asynchronous data transfer must either be direct unformatted, sequential unformatted, or stream unformatted. Asynchronous I/O to internal files is prohibited. Asynchronous I/O to raw character devices (for example, tapes or raw logical volumes) is prohibited. The *integer_variable* must not be associated with any entity in the data transfer I/O list, or with a *do_variable* of an *io_implied_do* in the data transfer I/O list. If the *integer_variable* is an array element reference, its subscript values must not be affected by the data transfer, the *io_implied_do* processing, or the definition or evaluation of any other specifier in the *io_control_spec*.

DELIM= *char_expr***(Fortran 2003)**

specifies what delimiter, if any, is used to delimit character constants written with list-directed or namelist formatting. *char_expr* is a scalar character expression whose value must evaluate to **APOSTROPHE**, **QUOTE**, or **NONE**. If the value is **APOSTROPHE**, apostrophes delimit character constants and all apostrophes within character constants are doubled. If the value is **QUOTE**, double quotation marks delimit character constants and all double quotation marks within character constants are doubled. If the value is **NONE**, character constants are not delimited and

no characters are doubled. The default value is **NONE**. The **DELIM=** specifier is permitted only for files being connected for formatted input/output, although it is ignored during input of a formatted record.

ERR= *stmt_label*
is an error specifier that specifies the statement label of an executable statement in the same scoping unit to which control is to transfer in the case of an error. Coding the **ERR=** specifier suppresses error messages.

NUM= *integer_variable***(IBM extension)**
is a number specifier that specifies the number of bytes of data transmitted between the I/O list and the file. *integer_variable* is an integer variable. The **NUM=** specifier is only permitted for unformatted output. Coding the **NUM** parameter suppresses the indication of an error that would occur if the number of bytes represented by the 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 output list items is not written into subsequent records. In the portion of the program that executes between the asynchronous data transfer statement and the matching **WAIT** statement, the *integer_variable* in the **NUM=** specifier or any variable associated with it must not be referenced, become defined, or become undefined.

[NML=] *name*
is a namelist specifier that specifies the name of a namelist list 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, and the first item must be the unit specifier with **UNIT=** omitted. 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 output statement.

ADVANCE= *char_expr*
is an advance specifier that determines whether nonadvancing output occurs for this statement. *char_expr* is a character expression that must evaluate to **YES** or **NO**. If **NO** is specified, nonadvancing output occurs. If **YES** is specified, advancing, formatted sequential or formatted stream output occurs. The default value is **YES**. **ADVANCE=** can be specified only in a formatted sequential **WRITE** statement with an explicit format specification that does not specify an internal file unit specifier.

Rules

IBM If a **NUM=** specifier is present, neither a format specifier nor a namelist specifier can be present. **IBM**

Variables specified for the **IOSTAT=** and **NUM=** specifiers must not be associated with any output list item, namelist list item, or **DO** variable of an implied-**DO** list. If such a specifier variable is an array element, its subscript values must not be affected by the data transfer, any implied-**DO** processing, or the definition or evaluation of any other specifier.

If the **ERR=** and **IOSTAT=** specifiers are set and an error is encountered during a synchronous data transfer, transfer is made to the statement specified by the **ERR=** specifier and a positive integer value is assigned to *ios*.

If the **ERR=** or **IOSTAT=** specifiers are set and an error is encountered during an asynchronous data transfer, execution of the matching **WAIT** statement is not required.

If a conversion error is encountered and the **CNVERR** run-time option is set to **NO**, **ERR=** is not branched to, although **IOSTAT=** may be set.

If **IOSTAT=** and **ERR=** are not specified,

- The program stops if a severe error is encountered.
- The program continues to the next statement if a recoverable error is encountered and the **ERR_RECOVERY** run-time option is set to **YES**. If the option is set to **NO**, the program stops.
- The program continues to the next statement when a conversion error is encountered if the **ERR_RECOVERY** run-time option is set to **YES**. If the **CNVERR** run-time option is set to **YES**, conversion errors are treated as recoverable errors; when **CNVERR=NO**, they are treated as conversion errors.

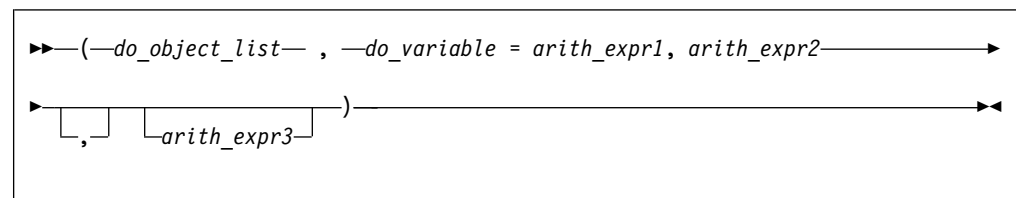`PRINT format` has the same effect as `WRITE(*,format)`.

Examples

```
WRITE (6,FMT='(10F8.2)') (LOG(A(I)),I=1,N+9,K),G
```

Related information

- "Asynchronous Input/Output" on page 220
- *Implementation details of XL Fortran Input/Output* in the *XL Fortran Optimization and Programming Guide*
- "Conditions and IOSTAT values" on page 226
- Chapter 9, "XL Fortran Input/Output," on page 215
- "READ" on page 441
- "WAIT (Fortran 2003)" on page 491
- *Setting Run-Time Options* in the *XL Fortran Compiler Reference*
- "Deleted features" on page 1054

Implied-DO List

```
►►──(──do_object_list── , ──do_variable = arith_expr1, arith_expr2───────────►

►──┬────────┬──)──────────────────────────────────────────►◄
   └──,──┴─arith_expr3─┘
```

do_object
      is an output list item

do_variable
      is a named scalar variable of type integer or real

arith_expr1, *arith_expr2*, **and** *arith_expr3*
      are scalar numeric expressions

The range of an implied-**DO** list is the list *do_object_list*. The iteration count and values of the **DO** variable are established from *arith_expr1*, *arith_expr2*, and *arith_expr3*, the same as for a **DO** statement. 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.

Chapter 12. Directives (IBM extension)

This section provides an alphabetical reference to non-SMP directives that apply to all platforms. For a complete listing and description of SMP and thread-safe directives, see the detailed directive descriptions in the *XL Fortran Optimization and Programming Guide*. For a detailed description of directives exclusive to the PowerPC® platform, see Chapter 13, "Hardware-specific directives," on page 545. This section contains the following contents:

Comment and noncomment form directives

XL Fortran directives belong to one of two groups: comment form directives and noncomment form directives.

Comment form directives

This section describes the format of comment form directives. The non-SMP comment form directives are as follows:

- **COLLAPSE**
- **SNAPSHOT**
- **SOURCEFORM**
- **SUBSCRIPTORDER**

Additional comment form directives can be found in "Directives and optimization" on page 506.

Format

▶▶—*trigger_head*—*trigger_constant*—*directive*————————————▶◀

trigger_head
    is one of **!**, *****, **C**, or **c** for fixed source form and **!** for free source form.

trigger_constant
    is **IBM*** by default.

Rules

The compiler treats all comment form directives, with the exception of those using the default *trigger_constant*, as comments, unless you define the appropriate *trigger_constant* using the **-qdirective** compiler option. As a result, code containing these directives is portable to non-SMP environments.

XL Fortran supports the OpenMP specification, as understood and interpreted by IBM. To ensure the greatest portability of code, we recommend that you use these directives whenever possible. You should use them with the OpenMP *trigger_constant*, **$OMP**; but you should not use this *trigger_constant* with any other directive.

XL Fortran also includes the *trigger_constant*s **IBMP** and **IBMT**. The rules are as follows:

- The compiler recognizes **IBMP** if you compile using the **-qsmp** compiler option. You should use **IBMP** with the **SCHEDULE** directive, and IBM extensions to OpenMP directives.
- The compiler recognizes **IBMT** if you compile using the **-qthreaded** compiler option. **IBMT** is the default for the **xlf_r**, **xlf90_r**, **xlf95_r**, **xlf2003_r**, **xlf2008_r**, and ⟨CUDA Fortran⟩ **xlcuf** ⟨CUDA Fortran⟩ invocation commands; we recommend its use with the **THREADLOCAL** directive.

The **-qsmp** compiler option implies **-qdirective=SMP\$:\$OMP:IBMP**, which turns on the **SMP\$**, **\$OMP**, and **IBMP** trigger constants, in addition to the default constant trigger **IBM***. You can specify an alternate or additional *trigger_constant* with the **-qdirective** compiler option.

XL Fortran directives include directives that are common to other vendors. If you use these directives in your code, you can enable whichever *trigger_constant* that vendor has selected. Specifying the trigger constant by using the **-qdirective** compiler option will enable the *trigger_constant* the vendor has selected.

The *trigger_head* follows the rules of comment lines either in Fortran 90 free source form or fixed source form. If the *trigger_head* is **!**, it does not have to be in column 1. There must be no blanks between the *trigger_head* and the *trigger_constant*.

You can specify the *directive_trigger* (defined as the *trigger_head* combined with the *trigger_constant*, **!IBM*** for example) and any directive keywords in uppercase, lowercase, or mixed case.

You can specify inline comments on directive lines.

```
!IBM* INDEPENDENT, NEW(i)    !This is a comment
```

A directive cannot follow another statement or another directive on the same line.

All comment form directives can be continued. You cannot embed a directive within a continued statement, nor can you embed a statement within a continued directive.

You must specify the *directive_trigger* on all continuation lines. However, the *directive_trigger* on a continuation line need not be identical to the *directive_trigger* that is used in the continued line. For example:

```
!IBM* INDEPENDENT &
!TRIGGER&  , REDUCTION (X)                &
!IBM*&  , NEW (I)
```

The above is equivalent to:

```
!IBM* INDEPENDENT, REDUCTION (X), NEW (I)
```

provided both **IBM*** and **TRIGGER** are active *trigger_constant*s.

For more information, see "Lines and source formats" on page 10.

You can specify a directive as a free source form or fixed source form comment, depending on the current source form.

Fixed source form rules

If the *trigger_head* is one of **C**, **c**, or *****, it must be in column 1.

The maximum length of the *trigger_constant* in fixed source form is 4 for directives that are continued on one or more lines. This rule applies to the continued lines only, not to the initial line. Otherwise, the maximum length of the *trigger_constant* is 15. We recommend that initial line triggers have a maximum length of 4. The maximum allowable length of 15 is permitted for the purposes of backwards compatibility.

If the *trigger_constant* has a length of 4 or less, the first line of a comment directive must have either white space or a zero in column 6. Otherwise, the character in column 6 is part of the *trigger_constant*.

The *directive_trigger* of a continuation line of a comment directive must appear in columns 1-5. Column 6 of a continuation line must have a character that is neither white space nor a zero.

For more information, see "Fixed source form" on page 11.

Free source form rules

The *trigger_head* is !. The maximum length of the *trigger_constant* is 15.

An ampersand (&) at the end of a line indicates that the directive will continue. When you continue a directive line, a *directive_trigger* must appear at the beginning of all continuation lines. If you are beginning a continuation line with an ampersand, the *directive_trigger* must precede the ampersand. For example:

```
!IBM* INDEPENDENT &
!IBM*&  , REDUCTION (X)        &
!IBM*&  , NEW (I)
```

For more information, see "Free source form" on page 13.

    Related information in the *XL Fortran Compiler Reference*

    📄 **-qdirective**

Noncomment form directives

This section describes the format of noncomment form directives, which include the following directives:
- **EJECT**
- **INCLUDE**
- **#LINE**
- **@PROCESS**

Format

```
►►──directive───────────────────────────────────────►◄
```

Rules

The compiler always recognizes noncomment form directives.

Noncomment form directives cannot be continued.

Additional statements cannot be included on the same line as a directive.

Source format rules concerning white space apply to directive lines.

Directives and optimization

The following directives are comment form directives useful for optimizing programs. See *Optimizing your applications* in the *XL Fortran Optimization and Programming Guide* and the compiler options that affect performance.

Assertive directives

Assertive directives gather information about source code that is otherwise unavailable to the compiler. Providing this information can increase performance.

- **ASSERT**
- **CNCALL**
- **EXECUTION_FREQUENCY**
- **EXPECTED_VALUE**
- **INDEPENDENT**
- **MEM_DELAY**
- **PERMUTATION**

Directives for loop optimization

The following directives provide different methods for loop optimization:

- **BLOCK_LOOP**
- **LOOPID**
- **SIMD_PEEL**
- **STREAM_UNROLL**
- **UNROLL**
- **UNROLL_AND_FUSE**

Detailed directive descriptions

ALIGN

Purpose

You can use the **ALIGN** directive to specify the alignment of variables in memory. This can improve performance when the alignment allows the use of the vector facilities.

Syntax

```
►►──ALIGN──(──alignment_boundary──,──var_list──)────────────────►◄
```

alignment_boundary
> The alignment boundary in bytes. It must be a constant scalar integer expression whose value is a power of 2, in the range 1 - 1, 048, 576.

var_list
> A comma-separated list of variable names to align on the specified boundary.

Rules

You can use the **ALIGN** directive only in the specification part of a compilation unit. In addition, the directive must be in the same scoping unit in which *var* is declared.

The **ALIGN** directive can only specify the alignment of variables. Therefore, the **ALIGN** directive must not be specified in derived type declarations.

You must not specify the same variable name in multiple **ALIGN** directives.

The **ALIGN** directive must not be specified for the following objects:
- Subobjects
- Use-associated variables
- Host-associated variables
- Record structures
- Common block names and objects
- Dummy arguments
- Named constants
- Variables that have VECTOR types
- Variables that are part of an equivalence group

The **ALIGN** directive has precedence over the **-qalign** option.

Examples

Example 1:

In the following example, the optimizer can use SIMD instructions to do the array add operation:

```
REAL x(4), y(4), z(4)
!IBM* ALIGN(16, x, y, z)
! Code to initialize x and y
z = x + y
END
```

Example 2:

In the following example, specifying the alignment of an allocatable variable determines the alignment of the target data.

```
REAL, ALLOCATABLE :: x(:)
!IBM* ALIGN(16, x)
ALLOCATE(x(20))
PRINT *, MOD(LOC(x(1)), 16) ! Prints 0
END
```

Related information

- **-qalign** option in the *XL Fortran Compiler Reference*

ASSERT

Purpose

The **ASSERT** directive provides the compiler with the characteristics of **DO** loops that can assist in optimizing source code.

Syntax

```
►►──ASSERT──(──assertion_list──)──────────────────────────────────►◄
```

assertion

> **ITERCNT**(*n*), **MINITERCNT**(*n*), **MAXITERCNT**(*n*), or **NODEPS**. All assertions are not mutually exclusive. You can use at most one of each assertion for the same **DO** loop. *n* must be a positive, scalar, integer constant expression.

> **ITERCNT**(*n*)
>> specifies the expected number of iterations *(n)* for a given **DO** loop.

> **MINITERCNT**(*n*)
>> specifies the expected minimum number of iterations *(n)* for a given **DO** loop.

> **MAXITERCNT**(*n*)
>> specifies the expected maximum number of iterations *(n)* for a given **DO** loop.

> **NODEPS**
>> specifies that no loop-carried dependencies exist within a given **DO** loop.

>> **NODEPS** takes effect only when you specify the **-qhot** or **-qsmp** compiler options.

Rules

The first noncomment line (not including other directives) following the **ASSERT** directive must be a **DO** loop. This line cannot be an infinite **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** loop. The **ASSERT** directive applies only to the **DO** loop immediately following the directive, and not to any nested **DO** loops.

ITERCNT, **MINITERCNT**, and **MAXITERCNT** are not required to be accurate. The values only affect performance, never correctness. Specify the values following the rule **MINITERCNT <= ITERCNT <= MAXITERCNT**. Otherwise, messages are issued to indicate that the values are inconsistent and the inconsistent value is

ignored. The assert directives **ITERCNT**, **MINITERCNT**, and **MAXITERCNT** take priority over the options specified with **-qassert={itercnt, minitercnt, maxitercnt}** for the given loop.

When **NODEPS** is specified, the user is explicitly declaring to the compiler that no loop-carried dependencies exist within the **DO** loop or any procedures invoked from within the **DO** loop. A loop-carried dependency involves two iterations within a **DO** loop interfering with one another. Interference occurs in the following situations:

- Two operations that define, undefine, or redefine the same atomic object (data that has no subobjects) interfere.
- Definition, undefinition, or redefinition of an atomic object interferes with any use of the value of the object.
- Any operation that causes the association status of a pointer to become defined or undefined interferes with any reference to the pointer or any other operation that causes the association status to become defined or undefined.
- Transfer of control outside the **DO** loop or execution of an **EXIT**, **STOP**, or **PAUSE** statement interferes with all other iterations.
- Any two input/output (I/O) operations associated with the same file or external unit interfere with each other. The exceptions to this rule are:
  - If the two I/O operations are two **INQUIRE** statements; or
  - F2003 If the two I/O operations are accessing distinct areas of a stream access file; or F2003
  - If the two I/O operations are accessing distinct records of a direct access file.
- A change in the allocation status of an allocatable object between iterations causes interference.

It is possible for two complementary **ASSERT** directives to apply to any given **DO** loop. However, an **ASSERT** directive cannot be followed by a contradicting **ASSERT** directive for a given **DO** loop:

```
!IBM* ASSERT (ITERCNT(10))
!IBM* INDEPENDENT, REDUCTION (A)
!IBM* ASSERT (ITERCNT(20))     ! invalid
      DO I = 1, N
          A(I) = A(I) * I
      END DO
```

In the example above, the **ASSERT(ITERCNT(20))** directive contradicts the **ASSERT(ITERCNT(10))** directive and is invalid.

The **ASSERT** directive overrides the **-qassert** compiler option for the **DO** loop on which the **ASSERT** directive is specified.

Examples

Example 1:

```
! An example of the ASSERT directive with NODEPS.
PROGRAM EX1
  INTEGER A(100)
!IBM* ASSERT (NODEPS)
  DO I = 1, 100
    A(I) = A(I) * FNC1(I)
  END DO
END PROGRAM EX1
```

```
FUNCTION FNC1(I)
  FNC1 = I * I
END FUNCTION FNC1
```

Example 2:
```
! An example of the ASSERT directive with NODEPS and ITERCNT.
SUBROUTINE SUB2 (N)
  INTEGER A(N)
!IBM*  ASSERT (NODEPS,ITERCNT(100))
  DO I = 1, N
    A(I) = A(I) * FNC2(I)
  END DO
END SUBROUTINE SUB2

FUNCTION FNC2 (I)
  FNC2 = I * I
END FUNCTION FNC2
```

Example 3:
```
! An example of the ASSERT directive with ITERCNT, MINITERCNT, and MAXITERCNT.
!IBM* ASSERT (ITERCNT(10), MINITERCNT(5))
DO I = 1, N
  A(I) = A(I) * I
!IBM* ASSERT (ITERCNT(100))
!IBM* ASSERT (MINITERCNT(5), MAXITERCNT(100))
  DO J = 1, M
    B(J) = A(I) + B(J)
  END DO
END DO
```

Related information
- **-qassert** option in the *XL Fortran Compiler Reference*
- **-qdirective** in the *XL Fortran Compiler Reference*
- "Loop parallelization" on page 374

BLOCK_LOOP
Purpose

The **BLOCK_LOOP** directive allows you to exert greater control over optimizations on a specific **DO** loop inside a loop nest. Using a technique called blocking, the **BLOCK_LOOP** directive separates large iteration count **DO** loops into smaller iteration groups. Execution of these smaller groups can increase the efficiency of cache space use and augment performance.

Applying **BLOCK_LOOP** to a loop with dependencies, or a loop with alternate entry or exit points will produce unexpected results.

The **BLOCK_LOOP** directive takes effect only when the **-qhot**, **-qipa**, or **-qsmp** compiler option is specified.

Syntax

```
►►──BLOCK_LOOP──(──n─────────────────────)──────────────────────────►◄
                     └─,──name_list─┘
```

n   is a positive integer expression as the size of the iteration group.

name   a unique identifier in the same scoping unit as **BLOCK_LOOP**, that you can create using the **LOOPID** directive.

If you do not specify *name*, blocking occurs on the first **DO** loop immediately following the **BLOCK_LOOP** directive.

Rules

For loop blocking to occur, a **BLOCK_LOOP** directive must immediately precede a **DO** loop.

You must not specify the **BLOCK_LOOP** directive more than once.

You must not specify the **BLOCK_LOOP** directive for a **DO WHILE** loop or an infinite **DO** loop.

Examples

```
! Loop Tiling for Multi-level Memory Heirarchy
          INTEGER :: M, N, i, j, k
          M = 1000
          N = 1000

    !IBM* BLOCK_LOOP(L3_cache_size, L3_cache_block)
          do i = 1, N

    !IBM* LOOPID(L3_cache_block)
    !IBM* BLOCK_LOOP(L2_cache_size, L2_cache_block)
            do j = 1, N

    !IBM* LOOPID(L2_cache_block)
              do k = 1, M
                do l = 1, M
                  .
                  .
                  .
                end do
              end do
            end do
          end do

          end

    ! The compiler generated code would be equivalent to:

        do index1 = 1, M, L3_cache_size
          do i = 1, N
            do index2 = index1, min(index1 + L3_cache_size, M), L2_cache_size
              do j = 1, N
                do k = index2, min(index2 + L2_cache_size, M)
                  do l = 1, M
                    .
                    .
                    .
                  end do
                end do
              end do
            end do
          end do
        end do
```

Related information
- For additional methods of optimizing loops, see the **STREAM UNROLL** and the **UNROLL** and **UNROLL_AND_FUSE** directives.
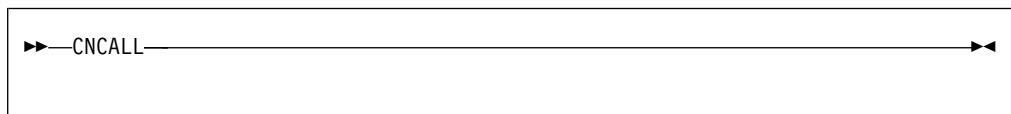
CNCALL

Purpose

When the **CNCALL** directive is placed before a **DO** loop, you are explicitly declaring to the compiler that no loop-carried dependencies exist within any procedure called from the **DO** loop.

This directive only takes effect if you specify the **-qsmp** or **-qhot** compiler option.

Syntax

```
►►──CNCALL──────────────────────────────────────────────────────────►◄
```

Rules

The first noncomment line (not including other directives) that is following the **CNCALL** directive must be a **DO** loop. This line cannot be an infinite **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** loop. The **CNCALL** directive applies only to the **DO** loop that is immediately following the directive and not to any nested **DO** loops.

When specifying the **CNCALL** directive, you are explicitly declaring to the compiler that no procedures invoked within the **DO** loop have any loop-carried dependencies. If the **DO** loop invokes a procedure, separate iterations of the loop must be able to concurrently call that procedure. The **CNCALL** directive does not assert that other operations in the loop do not have dependencies, it is only an assertion about procedure references.

A loop-carried dependency occurs when two iterations within a **DO** loop interfere with one another. See the **ASSERT** directive for the definition of interference.

Examples

```
! An example of CNCALL where the procedure invoked has
! no loop-carried dependency but the code within the
! DO loop itself has a loop-carried dependency.
      PROGRAM EX3
        INTEGER A(100)
  !IBM*  CNCALL
        DO I = 1, N
          A(I) = A(I) * FNC3(I)
          A(I) = A(I) + A(I-1)    ! This has loop-carried dependency
        END DO
      END PROGRAM EX3

      FUNCTION FNC3 (I)
        FNC3 = I * I
      END FUNCTION FNC3
```

Related information
- "INDEPENDENT" on page 521
- **-qdirective** in the *XL Fortran Compiler Reference*
- **-qhot** in the *XL Fortran Compiler Reference*
- **-qsmp** compiler option in the *XL Fortran Compiler Reference*
- "DO" on page 336
- "Loop parallelization" on page 374

COLLAPSE

Purpose

The **COLLAPSE** directive reduces an entire array dimension to a single element by specifying that only the element in the lower bound of an array dimension is accessible. If you do not specify a lower bound, the default lower bound is one.

Used with discretion, the **COLLAPSE** directive can facilitate an increase in performance by reducing repetitive memory access associated with multiple-dimension arrays.

Syntax

```
►►──COLLAPSE──(──collapse_array_list──)──────────────────────────────────────►◄
```

where *collapse_array* is:

```
►►───array_name──(──expression_list──)───────────────────────────────────────►◄
```

where *expression_list* is a comma separated list of *expression*.

array name
> is the array name.

expression
> is a constant scalar integer expression. You may only specify positive integer values.

Rules

The **COLLAPSE** directive must contain at least one array.

The **COLLAPSE** directive applies only to the scoping unit in which it is specified. The declarations of arrays contained in a **COLLAPSE** directive must appear in the same scoping unit as the directive. An array that is accessible in a scoping unit by use or host association must not be specified in a **COLLAPSE** directive in that scoping unit.

The lowest value you can specify in *expression_list* is one. The highest value must not be greater than the number of dimensions in the corresponding array.

A single scoping unit can contain multiple **COLLAPSE** declarations, though you can only specify an array once for a particular scoping unit.

You can not specify an array in both a **COLLAPSE** directive and an **EQUIVALENCE** statement.

You can not use the **COLLAPSE** directive with arrays that are components of derived types.

If you apply both the **COLLAPSE** and **SUBSCRIPTORDER** directives to an array, you must specify the **SUBSCRIPTORDER** directive first.

The **COLLAPSE** directive applies to:
- Assumed-shape arrays in which all lower bounds must be constant expressions.
- Explicit-shape arrays in which all lower bounds must be constant expressions.

Examples

Example 1: In the following example, the **COLLAPSE** directive is applied to the explicit-shape arrays *A* and *B*. Referencing *A(m,2:100,2:100)* and *B(m,2:100,2:100)* in the inner loops, become *A(m,1,1)* and *B(m,1,1)*.

```
!IBM* COLLAPSE(A(2,3),B(2,3))
      REAL*8 A(5,100,100), B(5,100,100), c(5,100,100)

      DO I=1,100
       DO J=1,100
        DO M=1,5
           A(M,J,I) = SIN(C(M,J,I))
           B(M,J,I) = COS(C(M,J,I))
        END DO
        DO M=1,5
         DO N=1,M
            C(M,J,I) = C(M,J,I) + A(N,J,I)*B(6-N,J,I)
         END DO
        END DO
       END DO
      END DO
      END
```

Related information

For more information on the **SUBSCRIPTORDER** directive, see

EJECT

Purpose

EJECT directs the compiler to start a new full page of the source listing. If there has been no source listing requested, the compiler will ignore this directive.

Syntax

```
►►──EJECT───────────────────────────────────────────────►◄
```
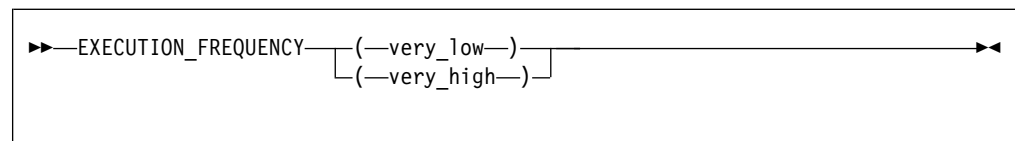
Rules

The **EJECT** compiler directive can have an inline comment and a label. However, if you specify a statement label, the compiler discards it. Therefore, you must not reference any label on an **EJECT** directive. An example of using the directive would be placing it before a **DO** loop that you do not want split across pages in the listing. If you send the source listing to a printer, the **EJECT** directive provides a page break.

EXECUTION_FREQUENCY (IBM extension)
Purpose

The **EXECUTION_FREQUENCY** directive marks source code that you expect will be executed very frequently or very infrequently.

Syntax

```
►►──EXECUTION_FREQUENCY──┬──(──very_low──)──┬──────────────────────────►◄
                         └──(──very_high──)─┘
```

Rules

The **EXECUTION_FREQUENCY** directive is a hint to the optimizer and only takes effect if optimization is selected.

EXECUTION_FREQUENCY is most effective within an execution control construct such as **IF**, **SELECT CASE**, and **SELECT TYPE**, and for labeled branch target statements.

EXECUTION_FREQUENCY should be the first statement within a control construct. If there are multiple **EXECUTION_FREQUENCY** directives in the same branch, only the first **EXECUTION_FREQUENCY** directive to have effect is used, the rest are ignored.

Examples

```
! An example of EXECUTION_FREQUENCY in an IF statement
integer function get_grade(student_id)
  integer student_id
  if (is_valid(student_id)) then
    ! get_grade is most often called with
    ! valid student_id's.
    !IBM* EXECUTION_FREQUENCY(VERY_HIGH)
    !...
  else
    ! We have an error.
  endif
end function
```

EXPECTED_VALUE

Purpose

Specifies the value that a dummy argument is most likely to have at run time. The compiler can use this information to perform certain optimizations, such as procedure cloning and inlining.

Syntax

```
►►──EXPECTED_VALUE──(──argument──,──value──)──────────────────────────────►◄
```

argument
    The name of the dummy argument for which you want to provide the expected value. The dummy argument must be a scalar of REAL, INTEGER, LOGICAL, or BYTE type. It must not have the ALLOCATABLE or POINTER attribute.

value   A constant expression representing the value that the dummy argument is most likely to take at run time.

Rules

Use the **EXPECTED_VALUE** directive only in the specification part of a procedure. Do not use more than one **EXPECTED_VALUE** directive for a dummy argument.

Examples

In the following example, the **EXPECTED_VALUE** directives indicate to the compiler that the most likely value is 1 for a, and 0 for b.

```
integer function func(a, b)
  integer a, b
  !IBM* EXPECTED_VALUE(a, 1)
  !IBM* EXPECTED_VALUE(b, 0)
  ...
end function func
```

FUNCTRACE_XLF_CATCH

Purpose

The **FUNCTRACE_XLF_CATCH** directive specifies that the procedure whose declaration immediately follows the directive is a catch tracing subroutine.

Syntax

```
►►──FUNCTRACE_XLF_CATCH────────────────────────────────────────────────►◄
```

Rules

The catch tracing procedure must have the same characteristics as the following interface:

```
subroutine routine_name(procedure_name, file_name, line_number, id)
  use, intrinsic :: iso_c_binding
  character(*), intent(in) :: procedure_name
  character(*), intent(in) :: file_name
  integer(c_int), intent(in) :: line_number
  type(c_ptr), intent(inout) :: id
end subroutine
```

Related information

"FUNCTRACE_XLF_ENTER"

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace_xlf_catch** compiler option.

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace** compiler option.
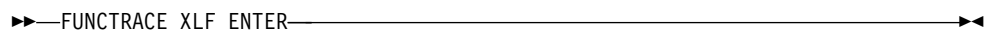
For detailed information about how to implement procedure tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

FUNCTRACE_XLF_ENTER
Purpose

The **FUNCTRACE_XLF_ENTER** directive specifies that the procedure whose declaration immediately follows the directive is an entry tracing subroutine.

Syntax

```
►►──FUNCTRACE_XLF_ENTER──────────────────────────────────────────►◄
```

Rules

The entry tracing procedure must have the same characteristics as the following interface:

```
subroutine routine_name(procedure_name, file_name, line_number, id)
  use, intrinsic :: iso_c_binding
  character(*), intent(in) :: procedure_name
  character(*), intent(in) :: file_name
  integer(c_int), intent(in) :: line_number
  type(c_ptr), intent(inout) :: id
end subroutine
```

Related information

"FUNCTRACE_XLF_CATCH" on page 516

"FUNCTRACE_XLF_EXIT"

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace_xlf_enter** compiler option.

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace** compiler option.

For detailed information about how to implement procedure tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

FUNCTRACE_XLF_EXIT
Purpose

The **FUNCTRACE_XLF_EXIT** directive specifies that the procedure whose declaration immediately follows the directive is an exit tracing subroutine.

Syntax

```
►►──FUNCTRACE_XLF_EXIT──────────────────────────────────────────────────►◄
```

Rules

The exit tracing procedure must have the same characteristics as the following interface:

```
subroutine routine_name(procedure_name, file_name, line_number, id)
  use, intrinsic :: iso_c_binding
  character(*), intent(in) :: procedure_name
  character(*), intent(in) :: file_name
  integer(c_int), intent(in) :: line_number
  type(c_ptr), intent(inout) :: id
end subroutine
```

Related information

"FUNCTRACE_XLF_CATCH" on page 516

"FUNCTRACE_XLF_ENTER" on page 517

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace_xlf_exit** compiler option.

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace** compiler option.

For detailed information about how to implement procedure tracing routines in your code, as well as detailed examples and a list of rules for using them, see

**Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide.*

IGNORE_TKR (IBM extension)
Purpose

The **IGNORE_TKR** directive allows the compiler to ignore the type, kind, and rank of dummy arguments when checking the interfaces of specific procedure calls, and when checking and resolving generic interfaces.

IGNORE_TKR allows you to port code from other Fortran compilers that support the **IGNORE_TKR** directive.

Syntax

```
►►──IGNORE_TKR─────────────────────────────────────────────────►◄
                 └─dummy_args_list─┘
```
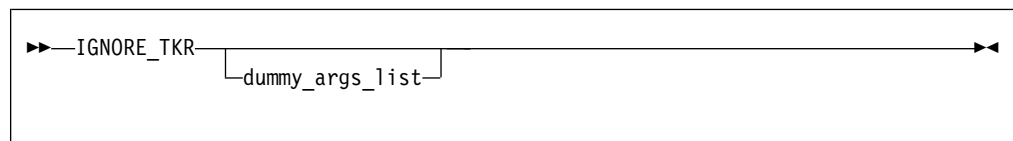
where *dummy_args_list* is a comma separated list of dummy argument names.

Rules

Dummy arguments that are allocatable, Fortran 90 pointers, assumed-shape arrays, or polymorphic must not be specified in the **IGNORE_TKR** directive.

I**IGNORE_TKR** may only appear in the body of an interface block and may specify dummy argument names only. **IGNORE_TKR** may appear before or after the declarations of the dummy arguments it specifies.

If dummy argument names are specified, **IGNORE_TKR** applies only those particular dummy arguments. If no dummy argument names are specified **IGNORE_TKR** applies to all dummy arguments except those that are allocatable objects, Fortran 90 pointers, assumed-shape arrays, or polymorphic entities.

Examples

```
interface
  subroutine sub1(a, b)
    integer(4) :: a
    integer(4) :: b
    !ibm* ignore_tkr b
  end subroutine
end interface

! valid calls
call sub1(1, 'abc')   ! type ignored
call sub1(1, 2_8)     ! kind ignored
call sub1(1, (/ 2 /)) ! rank ignored
```
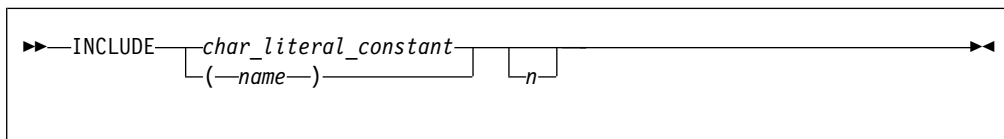
INCLUDE
Purpose

The **INCLUDE** compiler directive inserts a specified statement or a group of statements into a program unit.

Syntax

```
►►──INCLUDE──┬──char_literal_constant──┬──┬────┬──────────────►◄
             └──(──name──)─────────────┘  └─n─┘
```

name, char_literal_constant (delimiters are optional)
> specifies *filename*, the name of an include file
>
> You are not required to specify the full path of the wanted file, but must specify the file suffix if one exists.
>
> *name* must contain only characters allowable in the XL Fortran character set. See "Characters" on page 7 for the character set supported by XL Fortran.
>
> *char_literal_constant* is a character literal constant.

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, and cannot specify a kind type parameter. If you specify *n*, the compiler includes the file only if the number appears as a suboption in the **-qci** (conditional include) compiler option. If you do not specify *n*, the compiler always includes the file.

Conditional include allows you to selectively activate **INCLUDE** directives within Fortran source during compilation. Specify the files to include using the **-qci** compiler option.

In fixed source form, the **INCLUDE** compiler directive must start after column 6, and can have a label.

You can add an inline comment to the **INCLUDE** line.

Rules

An included file can contain any complete Fortran source statements and compiler directives, including other **INCLUDE** compiler directives. Recursive **INCLUDE** compiler directives are not allowed. An **END** statement can be part of the included group. The first and last included lines must not be continuation lines. The statements in the include file are processed with the source form of the including file.

If the **SOURCEFORM** directive appears in an include file, the source form reverts to that of the including file once processing of the include file is complete. After the inclusion of all groups, the resulting Fortran program must follow all of the Fortran rules for statement order.

For an **INCLUDE** compiler directive with the left and right parentheses syntax, XL Fortran translates the file name to lowercase unless the **-qmixed** compiler option is on.

The file system locates the specified *filename* as follows:
* If the first nonblank character of *filename* is **/**, *filename* specifies an absolute file name.

- If the first nonblank character is not /, the operating system searches directories in order of decreasing priority:
  - If you specify any **-I** compiler option, *filename* is searched for in the directories specified.
  - If the operating system cannot find *filename* then it searches:
    - the current directory for file *filename*.
    - the resident directory of the compiling source file for file *filename*.
    - directory /usr/include for file *filename*.

Examples

```
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 '../Abc'              ! includes file Abc from parent directory
                             ! of directory being searched
```

Related information

qci Option in the *XL Fortran Compiler Reference*

INDEPENDENT
Purpose

The **INDEPENDENT** directive, if used, must precede a **DO** loop, **FORALL** statement, or **FORALL** construct. It specifies that each operation in the **FORALL** statement or **FORALL** construct, can be executed in any order without affecting the semantics of the program. It also specifies that each iteration of the **DO** loop, can be executed in any order without affecting the semantics of the program.

Type

This directive only takes effect if you specify the **-qsmp** or **-qhot** compiler option.

Syntax

```
►►──INDEPENDENT──┬─────────────────────────────────────┬──►◄
                 │  ◄─────────────────────────────────┐ │
                 └─┬─,──NEW──(──named_variable_list──)──┬─┘
                   └─,──REDUCTION──(──named_variable_list──)─┘
```
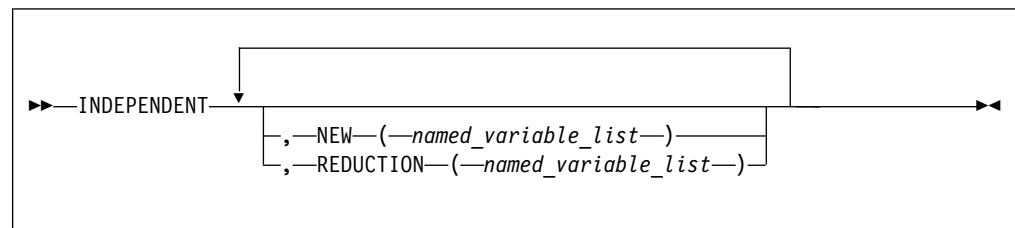
Rules

The first noncomment line (not including other directives) following the **INDEPENDENT** directive must be a **DO** loop, **FORALL** statement, or the first statement of a **FORALL** construct. This line cannot be an infinite **DO** or **DO WHILE** loop. The **INDEPENDENT** directive applies only to the **DO** loop that is immediately following the directive and not to any nested **DO** loops.

An **INDEPENDENT** directive can have at most one **NEW** clause and at most one **REDUCTION** clause.

If the directive applies to a **DO** loop, no iteration of the loop can interfere with any other iteration. Interference occurs in the following situations:

- Two operations that define, undefine, or redefine the same atomic object (data that has no subobjects) interfere, unless the parent object appears in the **NEW** clause or **REDUCTION** clause. You must define nested **DO** loop index variables in the **NEW** clause.
- Definition, undefinition, or redefinition of an atomic object interferes with any use of the value of the object. The exception is if the parent object appeared in the **NEW** clause or **REDUCTION** clause.
- Any operation that causes the association status of a pointer to become defined or undefined interferes with any reference to the pointer or any other operation that causes the association status to become defined or undefined.
- Transfer of control outside the **DO** loop or execution of an **EXIT**, **STOP**, or **PAUSE** statement interferes with all other iterations.
- If any two I/O operations associated with the same file or external unit interfere with each other. The exceptions to this rule are:
  - If the two I/O operations are two **INQUIRE** statements; or
  - F2003 If the two I/O operations are accessing distinct areas of a stream access file; or F2003
  - If the two I/O operations are accessing distinct records of a direct access file.
- A change in the allocation status of an allocatable object between iterations causes interference.

If the **NEW** clause is specified, the directive must apply to a **DO** loop. The **NEW** clause modifies the directive and any surrounding **INDEPENDENT** directives by accepting any assertions made by such directive(s) as true. It does this *even if* the variables specified in the **NEW** clause are modified by each iteration of the loop. Variables specified in the **NEW** clause behave as if they are private to the body of the **DO** loop. That is, the program is unaffected if these variables (and any variables associated with them) were to become undefined both before and after each iteration of the loop.

Any variable you specify in the **NEW** clause or **REDUCTION** clause must not:
- Be a dummy argument
- Be a pointee
- Be use-associated or host-associated
- Be a common block variable
- Have either the **SAVE** or **STATIC** attribute
- Have either the **POINTER** or **TARGET** attribute
- Appear in an **EQUIVALENCE** statement

For **FORALL**, no combination of index values affected by the **INDEPENDENT** directive assigns to an atomic storage unit that is required by another combination. If a **DO** loop, **FORALL** statement, or **FORALL** construct all have the same body and each is preceded by an **INDEPENDENT** directive, they behave the same way.

The **REDUCTION** clause asserts that updates to named variables will occur within **REDUCTION** statements in the **INDEPENDENT** loop. Furthermore, the intermediate values of the **REDUCTION** variables are not used within the parallel

section, other than in the updates themselves. Thus, the value of the **REDUCTION** variable after the construct is the result of a reduction tree.
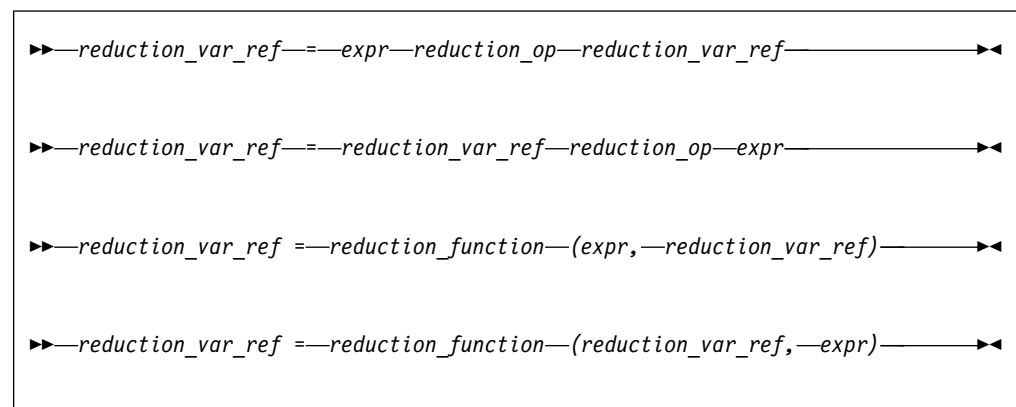
If you specify the **REDUCTION** clause, the directive must apply to a **DO** loop. The only reference to a **REDUCTION** variable in an **INDEPENDENT DO** loop must be within a reduction statement.

A **REDUCTION** variable must be of intrinsic type, but must not be of type character. A **REDUCTION** variable must not be an allocatable array.

A **REDUCTION** variable must not occur in:
- A **NEW** clause in the same **INDEPENDENT** directive
- A **NEW** or **REDUCTION** clause in an **INDEPENDENT** directive in the body of the following **DO** loop
- A **FIRSTPRIVATE**, **PRIVATE** or **LASTPRIVATE** clause in a **PARALLEL DO** directive in the body of the following **DO** loop
- A **PRIVATE** clause in a **PARALLEL SECTIONS** directive in the body of the following **DO** loop

A **REDUCTION** statement can have one of the following forms:

```
►►—reduction_var_ref—=—expr—reduction_op—reduction_var_ref——————————►◄


►►—reduction_var_ref—=—reduction_var_ref—reduction_op—expr——————————►◄


►►—reduction_var_ref =—reduction_function—(expr,—reduction_var_ref)————►◄


►►—reduction_var_ref =—reduction_function—(reduction_var_ref,—expr)————►◄
```

where:

reduction_var_ref
  is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op
  is one of: **+, -, *, .AND., .OR., .EQV., .NEQV.,** or **.XOR.**

reduction_function
  is one of: **MAX, MIN, IAND, IOR,** or **IEOR**

The following rules apply to **REDUCTION** statements:
1. A **REDUCTION** statement is an assignment statement that occurs in the range of an **INDEPENDENT DO** loop. A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the **INDEPENDENT DO** loop.
2. The two *reduction_var_ref*s that appear in a **REDUCTION** statement must be lexically identical.
3. The syntax of the **INDEPENDENT** directive does not allow you to designate an array element or array section as a **REDUCTION** variable in the

REDUCTION clause. Although such a subobject may occur in a REDUCTION
statement, it is the entire array that is treated as a REDUCTION variable.

4. You cannot use the following form of the REDUCTION statement:

```
►►──reduction_var_ref── = ──expr── - ──reduction_var_ref─────────────────►◄
```

Examples

Example 1:
```
      INTEGER A(10),B(10,12),F
!IBM*  INDEPENDENT                    ! The NEW clause cannot be
      FORALL (I=1:9:2) A(I)=A(I+1)    ! specified before a FORALL
!IBM*  INDEPENDENT, NEW(J)
      DO M=1,10
        J=F(M)                        ! 'J' is used as a scratch
        A(M)=J*J                      ! variable in the loop
!IBM*    INDEPENDENT, NEW(N)
        DO N=1,12                     ! The first executable statement
          B(M,N)=M+N*N                ! following the INDEPENDENT must
        END DO                        ! be either a DO or FORALL
      END DO
      END
```

Example 2:
```
      X=0
!IBM*  INDEPENDENT, REDUCTION(X)
      DO J = 1, M
        X = X + J**2
      END DO
```

Example 3:
```
      INTEGER A(100), B(100, 100)
!IBM*  INDEPENDENT, REDUCTION(A), NEW(J)  ! Example showing an array used
      DO I=1,100                          ! for a reduction variable
        DO J=1, 100
          A(I)=A(I)+B(J, I)
        END DO
      END DO
```

Related information

- "Loop parallelization" on page 374
- "DO construct" on page 138
- "FORALL" on page 371
- **-qdirective** in the *XL Fortran Compiler Reference*
- **-qhot** in the *XL Fortran Compiler Reference*
- **-qsmp** compiler option in the *XL Fortran Compiler Reference*

#LINE

Purpose

The **#line** directive associates code that is created by cpp or any other Fortran
source code generator with input code created by the programmer. Because the
preprocessor may cause lines of code to be inserted or deleted, the **#line** directive
can be useful in error reporting and debugging because it identifies which lines in

the original source caused the preprocessor to generate the corresponding lines in the intermediate file.

Syntax

```
>>--#line--line_number---------------------------------------><
                      |               |
                      |--filename-----|
```
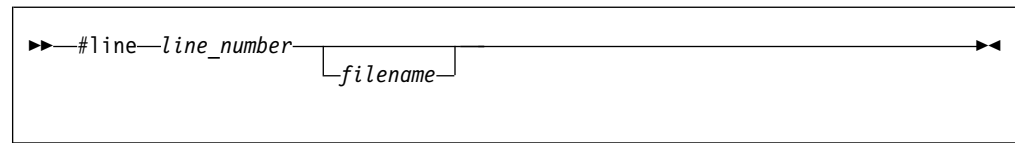
The **#line** directive is a noncomment directive and follows the syntax rules for this type of directive.

line_number
> is a positive, unsigned integer literal constant without a **KIND** parameter. You must specify *line_number*.

filename
> is a character literal constant, with no kind type parameter. The *filename* may specify a full or relative path. The *filename* as specified will be recorded for use later. If you specify a relative path, when you debug the program the debugger will use its directory search list to resolve the *filename*.

Rules

The **#line** directive follows the same rules as other noncomment directives, with the following exceptions:
- You cannot have inline comments on the same line as the **#line** directive.
- White space is optional between the **#** character and **line** in free source form.
- White space may not be embedded between the characters of the word **line** in fixed or free source forms.
- The **#line** directive can start anywhere on the line in fixed source form.

The **#line** directive indicates the origin of all code following the directive in the current file. Another **#line** directive will override a previous one.

If you supply a *filename*, the subsequent code in the current file will be as if it originated from that *filename*. If you omit the *filename*, and no previous **#line** directive with a specified *filename* exists in the current file, the code in the current file is treated as if it originated from the current file at the line number specified. If a previous **#line** directive with a specified *filename* does exist in the current file, the *filename* from the previous directive is used.

line_number indicates the position, in the appropriate file, of the line of code following the directive. Subsequent lines in that file are assumed to have a one to one correspondence with subsequent lines in the source file until another **#line** directive is specified or the file ends.

When XL Fortran invokes cpp for a file, the preprocessor will emit **#line** directives unless you also specify the **-d** option.

Examples

The file test.F contains:

```
!  File test.F, Line 1
#include "test.h"
PRINT*, "test.F Line 3"
...
PRINT*, "test.F Line 6"
#include "test.h"
PRINT*, "test.F Line 8"
END
```

The file test.h contains:

```
!  File test.h line 1
RRINT*,1              ! Syntax Error
PRINT*,2
```

After the C preprocessor processes the file test.F with the default options:

```
#line 1 "test.F"
! File test.F, Line 1
#line 1 "test.h"
! File test.h Line 1
RRINT*,1            ! Syntax Error
PRINT*,2
#line 3 "test.F"
PRINT*, "test.F Line 3"
...
#line 6
PRINT*, "test.F Line 6"
#line 1 "test.h"
!  File test.h Line 1
RRINT*,1            ! Syntax Error
PRINT*,2
#line 8 "test.F"
PRINT*, "test.F Line 8"
END
```

The compiler displays the following messages after it processes the file that is created by the C preprocessor:

```
2       2 |RRINT*,1
!Syntax error
          ......a...............
a - "test.h", line 2.6: 1515-019 (S) Syntax is incorrect.

4       2 |RRINT*,1             !Syntax error
          ......a...............
a - "test.h", line 2.6: 1515-019 (S) Syntax is incorrect.
```

Related information

- **-d** option in the *XL Fortran Compiler Reference*
- *Passing Fortran Files through the C Preprocessor* in the *XL Fortran Compiler Reference*

LOOPID

Purpose

The **LOOPID** directive allows you to assign a unique identifier to loop within a scoping unit. You can use the identifier to direct loop transformations. The **–qreport** compiler option can use the identifier you create to provide reports on loop transformations.

Syntax

```
►►──LOOPID──(──name──)──────────────────────────────────────►◄
```

name   is an identifier that must be unique within the scoping unit.

Rules

The **LOOPID** directive must immediately precede a **BLOCK_LOOP** directive or
DO construct.

You must not specify a **LOOPID** directive more than once for a given loop.

You must not specify a **LOOPID** directive for **DO** constructs without control
statements, **DO WHILE** constructs, `F2008` **DO CONCURRENT** constructs `F2008` ,
or an infinite **DO**.

Related information
* For additional methods of optimizing loops, see the **BLOCK_LOOP**, **STREAM
  UNROLL**, **UNROLL** and the **UNROLL_AND_FUSE** directives.

MEM_DELAY
Purpose

The **MEM_DELAY** directive specifies how many delay cycles there will be for
specific loads, these specific loads are delinquent loads with a long memory access
latency due to cache misses.

When you specify which load is delinquent the compiler may take that information
and carry out optimizations such as data prefetch.

Syntax

```
►►──MEM_DELAY──(──delinquent_variable──,──cycles──)──────────────►◄
```

delinquent_variable
          Any data item that can legally be passed by reference to a subprogram.

cycles   32-bit literal integer value or equivalent **PARAMETER**.

Rules

The **MEM_DELAY** directive is placed immediately before a statement which
contains a specified memory reference.

cycles must be a compile time constant, typically either L1 miss latency or L2 miss
latency.

Examples

```
program mem1
integer::i,n
integer::a(20),b(400)

n=20
do i=1,n
!IBM* mem_delay(b(n*i),10)
a(i)=b(n*i)
end do;
end
```

NEW

Purpose

Use the **NEW** directive to specify which variables should be local in a **PARALLEL DO** loop or a **PARALLEL SECTIONS** construct. This directive performs the same function as the **PRIVATE** clause of the **PARALLEL DO** directive and **PARALLEL SECTIONS** directive.

Class

The **NEW** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──NEW──named_variable_list─────────────────────────────────────────────►◄
```

Rules

The **NEW** directive must immediately follow either a **PARALLEL DO** directive or a **PARALLEL SECTIONS** directive.

If you specify the **NEW** directive, you must specify the corresponding **PARALLEL DO** or **PARALLEL SECTIONS** directive with no clauses.

If the **NEW** directive follows the **PARALLEL DO** directive, the first noncomment line (not including other directives) following the **NEW** directive must be a **DO** loop. This line cannot be an infinite **DO**, **F2008** **DO CONCURRENT** **F2008** , or **DO WHILE** loop.

A variable name in the *named_variable_list* of the **NEW** directive has the same restrictions as a variable name appearing in the **PRIVATE** clause of the **PARALLEL DO** directive or a **PRIVATE** clause of the **PARALLEL SECTIONS** directive. See the sections on the directive and the construct in the *XL Fortran Optimization and Programming Guide*.

Examples

```
INTEGER A(10), C(10)
REAL B(10)
INTEGER FUNC(100)
!SMP$ PARALLEL DO
!SMP$ NEW I, TMP
```

```
              DO I = 1, 10
                TMP = A(I) + COS(B(I))
                C(I) = TMP + FUNC(I)
              END DO
```

NOFUNCTRACE

Purpose

The **NOFUNCTRACE** directive disables tracing for the procedure or module whose declaration immediately follows the directive.

Syntax

```
▶▶──NOFUNCTRACE─────────────────────────────────────────────────────────▶◀
```

Rules

The **NOFUNCTRACE** directive must appear directly before the declaration of a procedure or a module. If the directive appears before a procedure declaration, it applies to the procedure and all its internal procedures. If the directive appears before a module declaration, it applies to all procedures in the module. The **NOFUNCTRACE** directive can appear immediately before internal procedure declarations.

Examples

The following example illustrates the use of the **NOFUNCTRACE** directive when you compile with **-qfunctrace**:

```
! None of the procedures in module m will be traced
!IBM* NOFUNCTRACE
MODULE M
CONTAINS
  SUBROUTINE modsub1
    CALL internal1
  CONTAINS
    SUBROUTINE internal1
    END SUBROUTINE internal1
  END SUBROUTINE modsub1
END MODULE M

MODULE n
CONTAINS
  ! modsub2 and its internal procedure internal3 will be traced.
  ! internal procedure internal2 will not be traced.
  SUBROUTINE modsub2
    CALL internal2
    CALL internal3
  CONTAINS
    !IBM* NOFUNCTRACE
    SUBROUTINE internal2
    END SUBROUTINE internal2

    SUBROUTINE internal3
    END SUBROUTINE internal3
  END SUBROUTINE modsub2

  ! modsub3 and its internal procedure internal4 will not be traced.
  !IBM* NOFUNCTRACE
```

```
      SUBROUTINE modsub3
        CALL internal4
      CONTAINS
        SUBROUTINE internal4
        END SUBROUTINE internal4
      END SUBROUTINE modsub3
END MODULE n

! The program and its internal procedure internal5 will not be traced.
!IBM* NOFUNCTRACE
PROGRAM nofunctrace
  USE m
  USE n
  CALL modsub1
  CALL modsub2
  CALL modsub3
  CALL internal5
CONTAINS
  SUBROUTINE internal5
  END SUBROUTINE internal5
END PROGRAM nofunctrace
```

Related information

See the *XL Fortran Compiler Reference* for details about the **-qfunctrace** compiler option.

NOSIMD
Purpose

The **NOSIMD** directive prohibits the compiler from automatically generating Vector Multimedia Extension (VMX) or Vector Scalar Extension (VSX) instructions in the loop immediately following the directive, or in the **FORALL** construct.

Syntax

```
►►─NOSIMD─────────────────────────────────────────────────────────────►◄
```
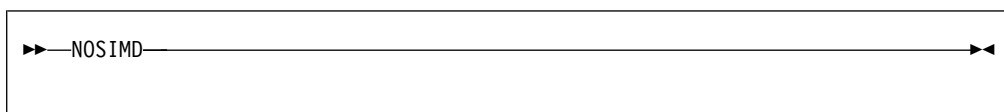
Rules

The first noncomment line (not including other directives) following the **NOSIMD** directive must be a **DO** loop, **FORALL** statement, or a **FORALL** construct. This line cannot be an infinite **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** loop. The **NOSIMD** directive applies only to the **DO** loop, **FORALL** statement, or the **FORALL** construct that is immediately following the directive and does not apply to any nested **DO** loops, nested **FORALL** statement or construct, or nested **DO** loops generated by the compiler for array language.

You can use the **NOSIMD** directive together with loop optimization and SMP directives.

Examples

```
      SUBROUTINE VEC (A, B)
        REAL*8 A(200), B(200)
        !IBM*  NOSIMD
        FORALL (N = 1:200), B(N) = B(N) / A(N)
      END SUBROUTINE
```

Related information

Refer to the compiler option for information on controlling Vector support for an entire application.

NOVECTOR
### Purpose

The **NOVECTOR** directive prohibits the compiler from auto-vectorizing the loop immediately following the directive. Auto-vectorization involves converting certain operations performed in a loop and on successive array elements into a call to a routine that computes several results simultaneously.

Syntax

```
►►──NOVECTOR────────────────────────────────────────────────────►◄
```

Rules

The first noncomment line (not including other directives) following the **NOVECTOR** directive must be a **DO** loop, **FORALL** statement, or a **FORALL** construct. This line cannot be an infinite **DO**, `F2008` **DO CONCURRENT** `F2008`, or **DO WHILE** loop. The **NOVECTOR** directive applies only to the **DO** loop, **FORALL** statement or the **FORALL** construct that is immediately following the directive and does not apply to any nested **DO** loops, or nested **FORALL** construct or statement.

You can use the **NOVECTOR** directive together with loop optimization and SMP directives.

Examples

```
      SUBROUTINE VEC (A, B)
        REAL*8 A(200), B(200)
        !IBM*  NOVECTOR
        DO N = 1, 200
          B(N) = B(N) / A(N)
        END DO
      END SUBROUTINE
```

Related information

Refer to the **-qhot=vector** compiler option for information on controlling auto-vectorization for an entire application.

PERMUTATION
Purpose

The **PERMUTATION** directive specifies that the elements of each array that is listed in the *integer_array_name_list* have no repeated values. This directive is useful when you use array elements as subscripts for other array references.

The **PERMUTATION** directive only takes effect if you specify the **-qsmp** or **-qhot** compiler option.

Syntax

```
►►──PERMUTATION──(──integer_array_name_list──)──────────────────────────────►◄
```
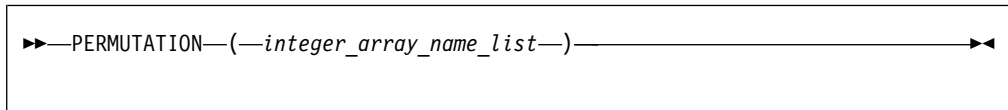
integer_array_name
> is an integer array with no repeated values.

Rules

The first noncomment line (not including other directives) that is following the **PERMUTATION** directive must be a **DO** loop. This line cannot be an infinite **DO**, F2008 **DO CONCURRENT** F2008 , or **DO WHILE** loop. The **PERMUTATION** directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

Examples

```
PROGRAM EX3
  INTEGER A(100), B(100)
  !IBM* PERMUTATION (A)
  DO I = 1, 100
    A(I) = I
    B(A(I)) = B(A(I)) + A(I)
  END DO
END PROGRAM EX3
```

Related information

- **-qhot** option in the *XL Fortran Compiler Reference*
- **-qsmp** compiler option in the *XL Fortran Compiler Reference*
- **DO**

@PROCESS
Purpose

The **@PROCESS** directive allows you to specify at the source level that a compiler option affects only an individual compilation unit. The directive can override options that you include in the configuration file, in the default settings, or on the command line. Refer to the *XL Fortran Compiler Reference* for information on limitations or restrictions for specifying a particular compiler option at the source level.

Syntax

```
          ┌──────────,──────────┐
          │                     │
►►──@PROCESS──▼──option──────────────────┬──────────────────────────────────►◄
                    └─(──suboption_list──)─┘
```
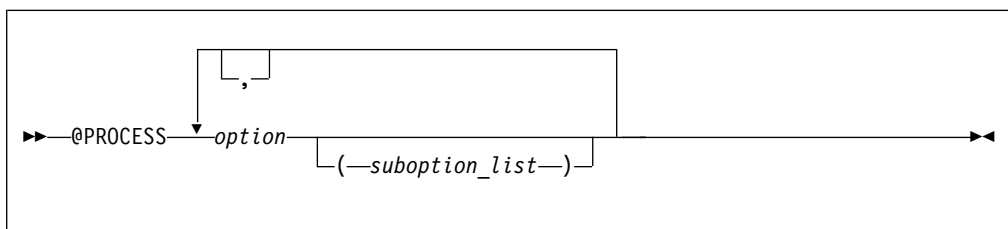
option   is the name of a compiler option, without **-q**

suboption
> is a suboption of a compiler option

Rules

In fixed source form, the **@PROCESS** directive can start in column 1 or after column 6. In free source form, the **@PROCESS** directive can start in any column.

You cannot place a statement label or inline comment on the same line as an **@PROCESS** compiler directive.

By default, any option settings you designate with the **@PROCESS** compiler directive are effective only for the compilation unit in which the statement appears. If the file has more than one compilation unit, the option returns to the original setting before compilation of the next unit. Trigger constants you specify using the **DIRECTIVE** option are in effect until the end of the file, or until processing **NODIRECTIVE**.

The **@PROCESS** compiler directive must appear before the first statement of a compilation unit. The only exceptions are for **SOURCE** and **NOSOURCE** compiler options, which you can specify in **@PROCESS** directives anywhere within the compilation unit.

Related information

See *Compiler Option Details* in the *XL Fortran Compiler Reference* for details on compiler options.

SIMD_PEEL

Purpose

The **SIMD_PEEL** directive instructs the compiler to peel a SIMDizable loop.

Syntax

array_name
> is the name of the array with respect to which the compiler peels a loop.

n   is a positive scalar integer constant expression.

Rules

When you specify *array_name* for **SIMD_PEEL**, the compiler peels a loop with respect to the first reference to the array inside the loop. The array references that have the same indexes as the first array reference are aligned. Other array references might be misaligned if they have different indexes than the first array reference.

When you specify *n* for **SIMD_PEEL**, the compiler peels a loop once for *n* iterations. When *n* is greater than the iteration number of the loop, the compiler

peels all the loop iterations only. Specifying *n* for **SIMD_PEEL** is useful especially
when the lower bounds of the induction variables of loops are constant.

Examples

The following example directs the compiler to peel the innermost loop with respect
to the first reference to array a inside that loop.

```
DO k = zmin, zmax
    DO j = ymin, ymax
        !IBM* SIMD_PEEL(a)
        DO i = xmin, xmax
            a(i,j,k) = b(i,j,k)*2
```

There are multiple references to array a with varying indexes in the following
example. The compiler peels the innermost loop with respect to a(i,j,k), which is
the first array reference inside that loop. The references to a(i,j,k) are aligned;
while the references to a(i+1,j,k) might be misaligned.

```
DO k = zmin, zmax
    DO j = ymin, ymax
        !IBM* SIMD_PEEL(a)
        DO i = xmin, xmax
            b(i,j,k) = a(i,j,k) - a(i+1,j,k)
            c(i,j,k) = a(i+1,j,k) + a(i,j,k)
```

In the following example, the compiler peels the innermost loop once for three
iterations.

```
DO k = 1, zmax
    DO j = 1, ymax
        !IBM* SIMD_PEEL(3)
        DO i = 1, xmax
            a(i,j,k) = b(i,j,k)*2
```

SNAPSHOT
Purpose

You can use the **SNAPSHOT** directive to specify a safe location where a
breakpoint can be set with a debug program, and provide a set of variables that
must remain visible to the debug program.

There can be a small reduction in performance at the point where the **SNAPSHOT**
directive is set, because the variables must be kept in memory for the debug
program to access. Variables made visible by the **SNAPSHOT** directive are
read-only. Undefined behavior will occur if these variables are modified through
the debugger. Use with discretion.

At high optimization levels, the **SNAPSHOT** directive does not consistently
preserve the contents of variables with a static storage class.

Syntax

```
►►──SNAPSHOT──(──named_variable_list──)──────────────────────────────────►◄
```

named_variable
>   is a named variable that must be accessible in the current scope.

Rules

To use the **SNAPSHOT** directive, you must specify the **-qdbg** compiler option at compilation.

Examples

Example 1: In the following example, the **SNAPSHOT** directive is used to monitor the value of private variables.

```
      INTEGER :: IDX
      INTEGER :: OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
      INTEGER, ALLOCATABLE :: ARR(:)
!      ...

!$OMP PARALLEL, PRIVATE(IDX)
!$OMP MASTER
      ALLOCATE(ARR(OMP_GET_NUM_THREADS()))
!$OMP END MASTER
!$OMP BARRIER

    IDX = OMP_GET_THREAD_NUM() + 1

!IBM* SNAPSHOT(IDX)                     ! The PRIVATE variable IDX is made visible
                                        ! to the debugger.
    ARR(IDX) = 2*IDX + 1

!$OMP END PARALLEL
```

Example 2: In the following example, the **SNAPSHOT** directive is used to monitor the intermediate values in debugging the program.

```
      SUBROUTINE SHUFFLE(NTH, XDAT)
        INTEGER, INTENT(IN) :: NTH
        REAL, INTENT(INOUT) :: XDAT(:)
        INTEGER :: I_TH, IDX, PART(1), I, J, LB, UB
        INTEGER :: OMP_GET_THREAD_NUM
        INTEGER(8) :: Y=1
        REAL :: TEMP

        CALL OMP_SET_NUM_THREADS(NTH)
        PART = UBOUND(XDAT)/NTH

!$OMP   PARALLEL, PRIVATE(NUM_TH, I, J, LB, UB, IDX, TEMP), SHARED(XDAT)
          NUM_TH = OMP_GET_THREAD_NUM() + 1
          LB = (NUM_TH - 1)*PART(1) + 1
          UB = NUM_TH*PART(1)

          DO I=LB, UB
!$OMP       CRITICAL
              Y = MOD(65539_8*y, 2_8**31)
              IDX = INT(REAL(Y)/REAL(2_8**31)*(UB - LB) + LB)

!SMP$         SNAPSHOT(i, y, idx, num_th, lb, ub)

!$OMP       END CRITICAL
            TEMP = XDAT(I)
            XDAT(I) = XDAT(IDX)
            XDAT(IDX) = TEMP
          ENDDO

!SMP$    SNAPSHOT(TEMP)                    ! The user can examine the value
```

```
                                        ! of the TEMP variable

     !$OMP   END PARALLEL
          END
```

Related information

See the *XL Fortran Compiler Reference* for details on the **-g** or **-qdbg** compiler option.

SOURCEFORM
Purpose

The **SOURCEFORM** compiler directive indicates that all subsequent lines are to be processed in the specified source form until the end of the file is reached or until an **@PROCESS** directive or another **SOURCEFORM** directive specifies a different source form.

Syntax

```
►►──SOURCEFORM──(──source──)─────────────────────────────────►◄
```

source is one of the following: **FIXED**, **FIXED**(*right_margin*), **FREE(F90)**, **FREE(IBM)**, or **FREE**. **FREE** defaults to **FREE(F90)**.

> *right_margin*
> is an unsigned integer specifying the column position of the right margin. The default is 72. The maximum is 132.

Rules

The **SOURCEFORM** directive can appear anywhere within a file. An include file is compiled with the source form of the including file. If the **SOURCEFORM** directive appears in an include file, the source form reverts to that of the including file once processing of the include file is complete.

The **SOURCEFORM** directive cannot specify a label.

Tip

To modify your existing files to Fortran 90 free source form where include files exist:

1. Convert your include files to Fortran 90 free source form: add a **SOURCEFORM** directive to the top of each include file. For example:

   ```
   !CONVERT*SOURCEFORM (FREE(F90))
   ```

   Define your own *trigger_constant* for this conversion process.

2. Once all the include files are converted, convert the .f files. Add the same **SOURCEFORM** directive to the top of each file, or ensure that the .f file is compiled with **-qfree=f90**.

3. Once all files have been converted, you can disable the processing of the directives with the **-qnodirective** compiler option. Ensure that **-qfree=f90** is used at compile time. You can also delete any unnecessary **SOURCEFORM** directives.

Examples

```
@PROCESS DIRECTIVE(CONVERT*)
      PROGRAM MAIN          ! Main program not yet converted
      A=1; B=2
      INCLUDE 'freeform.f'
      PRINT *, RESULT       ! Reverts to fixed form
      END
```

where file freeform.f contains:

```
!CONVERT* SOURCEFORM(FREE(F90))
RESULT = A + B
```

STREAM_UNROLL
Purpose

The **STREAM_UNROLL** directive instructs the compiler to apply the combined functionality of software prefetch and loop unrolling to **DO** loops with a large iteration count. Stream unrolling functionality optimizes **DO** loops to use multiple streams. You can specify the **STREAM_UNROLL** directive for both inner and outer **DO** loops, and the compiler will use an optimal number of streams to perform stream unrolling where applicable. Applying the **STREAM_UNROLL** directive to a loop with dependencies will produce unexpected results.
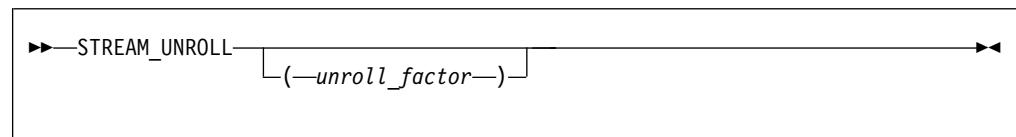
Syntax

```
►►──STREAM_UNROLL───────────────────────────────────►◄
                   └─(─unroll_factor─)─┘
```

unroll_factor

> The *unroll_factor* must be a positive scalar integer constant expression. An *unroll_factor* of 1 disables loop unrolling. If you do not specify an *unroll_factor*, the compiler determines the optimal number to perform stream unrolling.

Rules

You must specify one of the following compiler options to enable loop unrolling:
- **–O3** or higher optimization level

- **-qhot** compiler option
- **-qsmp** compiler option

Note that if the **-qstrict** option is in effect, no stream unrolling will occur. If you want to enable stream unrolling with the **-qhot** option alone, you must also specify **-qstrict=none**.

The **STREAM_UNROLL** directive must immediately precede a **DO** loop.

You must not specify the **STREAM_UNROLL** directive more than once, or combine the directive with **UNROLL**, **NOUNROLL**, **UNROLL_AND_FUSE**, or **NOUNROLL_AND_FUSE** directives for the same **DO** construct.

You must not specify the **STREAM_UNROLL** directive for a **DO WHILE** loop or an infinite **DO** loop.

Examples

The following is an example of how **STREAM_UNROLL** can increase performance.

```
      integer, dimension(1000) :: a, b, c
      integer i, m, n

!IBM* stream_unroll(4)
      do i =1, n
        a(i) = b(i) + c(i)
      enddo
      end
```

An *unroll factor* reduces the number of iterations from n to n/4, as follows:

```
m = n/4
do  i =1, n/4
    a(i) = b(i) + c(i)
    a(i+m) = b(i+m) + c(i+m)
    a(i+2*m) = b(i+2*m) + c(i+2*m)
    a(i+3*m) = b(i+3*m) + c(i+3*m)
enddo
```

The increased number of read and store operations are distributed among a number of streams determined by the compiler, reducing computation time and boosting performance.

Related information

- For further information on using prefetch techniques in XL Fortran see the **PREFETCH** directive set.
- For additional methods on optimizing loops, see the **BLOCK_LOOP**, **LOOPID**, **UNROLL** and the **UNROLL_AND FUSE** directives.

SUBSCRIPTORDER
Purpose

The **SUBSCRIPTORDER** directive rearranges the subscripts of an array. This results in a new array shape, since the directive changes the order of array dimensions in the declaration. All references to the array are correspondingly rearranged to match the new array shape.

Used with discretion, the **SUBSCRIPTORDER** directive may improve performance by increasing the number of cache hits and the amount of data prefetching. You may have to experiment with this directive until you find the arrangement that yields the most performance benefits. You may find **SUBSCRIPTORDER** especially useful when porting code originally intended for a non-cached hardware architecture.

In a cached hardware architecture, such as the PowerPC, an entire cache line of data is often loaded into the processor in order to access each data element. Changing the storage arrangement can be used to ensure that consecutively accessed elements are stored contiguously. This may result in a performance improvement, as there are more element accesses for each cache line referenced. Additionally, contiguous arrays which are consecutively accessed may help to better exploit the processor's prefetching facility.

Syntax

```
>>--SUBSCRIPTORDER--(----subscriptorder_array----)-------------><
                       |-- , <--|
```
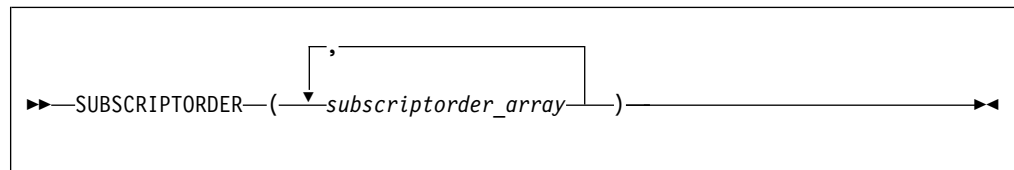
where *subscriptorder_array* is:

```
>>---array_name--(----subscriptorder_number----)--------------><
                    |-- , <--|
```
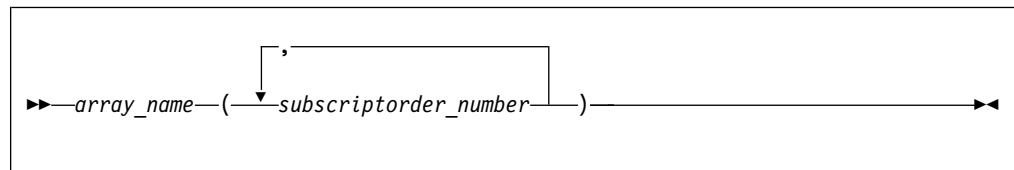
array name
        is the name of an array.

subscriptorder_number
        is an integer constant.

Rules

The **SUBSCRIPTORDER** directive must appear in a scoping unit preceding all declarations (after all **USE** statements) and references to the arrays in the *subscriptorder_array* list. The directive only applies to that scoping unit and must contain at least one array. If multiple scoping units share an array, then you must apply the **SUBSCRIPTORDER** directive to each of the applicable scoping units with identical subscript arrangements. Examples of methods of array sharing between scoping units include **COMMON** statements, **USE** statements, and subroutine arguments.

The lowest subscript number in a *subscriptorder_number* list must be 1. The highest number must be equal to the number of dimensions in the corresponding array. Every integer number between these two limits, including the limits, signifies a subscript number prior to rearrangement and must be included exactly once in the list.

You must not apply a **SUBSCRIPTORDER** directive multiple times to a particular array in a scoping unit.

You must maintain array shape conformance in passing arrays as actual arguments to elemental procedures, if one of the arrays appears in a **SUBSCRIPTORDER** directive. You must also adjust the actual arguments of the **SHAPE**, **SIZE**, **LBOUND**, and **UBOUND** inquiry intrinsic procedures and of most transformational intrinsic procedures.

You must manually modify data in input data files and in explicit initializations for arrays that appear in the **SUBSCRIPTORDER** directive.

On arrays to which the **COLLAPSE** directive is also applied, the **COLLAPSE** directive always refers to the pre-subscriptorder dimension numbers.

You must not rearrange the last dimension of an assumed-size array.

Examples

Example 1: In the following example, the **SUBSCRIPTORDER** directive is applied to an explicit-shape array and swaps the subscripts in every reference to the array, without affecting the program output.

```
!IBM* SUBSCRIPTORDER(A(2,1))
      INTEGER COUNT/1/, A(3,2)

      DO J = 1, 3
       DO K = 1, 2
    ! Inefficient coding: innermost index is accessing rightmost
    ! dimension.  The subscriptorder directive compensates by
    ! swapping the subscripts in the array's declaration and
    ! access statements.
    !
         A(J,K) = COUNT
         PRINT*, J, K, A(J,K)

         COUNT = COUNT + 1
       END DO
      END DO
```

Without the directive above, the array shape is (3,2) and the array elements would be stored in the following order:

```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)
```

With the directive, the array shape is (2,3) and the array elements are stored in the following order:

```
A(1,1) A(2,1) A(1,2) A(2,2) A(1,3) A(2,3)
```

Related information

For more information on the **COLLAPSE** directive, see "COLLAPSE" on page 513

UNROLL
Purpose

The **UNROLL** directive instructs the compiler to attempt loop unrolling where applicable. Loop unrolling replicates the body of the **DO** loop to reduce the number of iterations required to complete the loop.

You can control loop unrolling for an entire file using the **-qunroll** compiler
option. Specifying the directive for a particular **DO** loop always overrides the
compiler option.
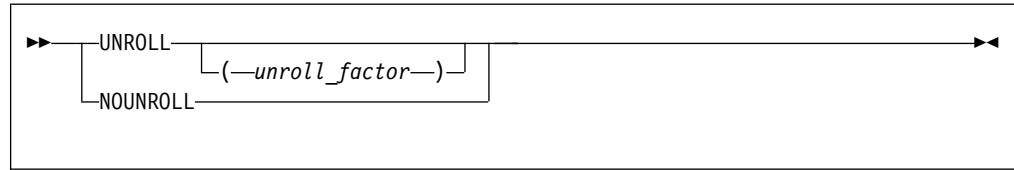
Syntax

```
►►──┬─UNROLL───────────────────────┬──────────────────►◄
    │         └─(─unroll_factor─)─┘ │
    └─NOUNROLL────────────────────┘
```

unroll_factor

> The *unroll_factor* must be a positive scalar integer constant expression. An
> *unroll_factor* of 1 disables loop unrolling. If you do not specify an
> *unroll_factor*, loop unrolling is compiler determined.

Rules

You must specify one of the following compiler options to enable loop unrolling:
- **–O3** or higher optimization level
- **-qhot** compiler option
- **-qsmp** compiler option

The **UNROLL** directive must immediately precede a **DO** loop.

You must not specify the **UNROLL** directive more than once, or combine the
directive with **NOUNROLL**, **STREAM_UNROLL**, **UNROLL_AND_FUSE**, or
NOUNROLL_AND_FUSE directives for the same **DO** construct.

You must not specify the **UNROLL** directive for a **DO WHILE** loop or an infinite
DO loop.

Examples

Example 1: In this example, the **UNROLL(2)** directive is used to tell the compiler
that the body of the loop can be replicated so that the work of two iterations is
performed in a single iteration. Instead of performing 1000 iterations, if the
compiler unrolls the loop, it will only perform 500 iterations.

```
!IBM* UNROLL(2)
      DO I = 1, 1000
         A(I) = I
      END DO
```

If the compiler chooses to unroll the previous loop, the compiler translates the loop
so that it is essentially equivalent to the following:

```
      DO I = 1, 1000, 2
         A(I) = I
         A(I+1) = I + 1
      END DO
```

Example 2: In the first **DO** loop, **UNROLL(3)** is used. If unrolling is performed,
the compiler will unroll the loop so that the work of three iterations is done in a
single iteration. In the second **DO** loop, the compiler determines how to unroll the
loop for maximum performance.

```
      PROGRAM GOODUNROLL

      INTEGER I, X(1000)
      REAL A, B, C, TEMP, Y(1000)

!IBM* UNROLL(3)
      DO I = 1, 1000
         X(I) = X(I) + 1
      END DO

!IBM* UNROLL
      DO I = 1, 1000
         A = -I
         B = I + 1
         C = I + 2
         TEMP = SQRT(B*B - 4*A*C)
         Y(I) = (-B + TEMP) / (2*A)
      END DO
      END PROGRAM GOODUNROLL
```

Related information
- For additional methods of optimizing loops, see the **BLOCK_LOOP**, **LOOPID**, **STREAM UNROLL** and the **UNROLL_AND_FUSE** directives.

UNROLL_AND_FUSE
Purpose

The **UNROLL_AND_FUSE** directive instructs the compiler to attempt a loop unroll and fuse where applicable. Loop unrolling replicates the body of multiple **DO** loops and combines the necessary iterations into a single unrolled loop. Using a fused loop can minimize the required number of loop iterations, while reducing the frequency of cache misses. Applying the **UNROLL_AND_FUSE** directive to a loop with dependencies will produce unexpected results.
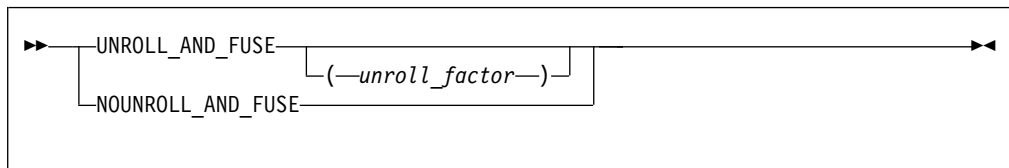
Syntax

```
>>─────UNROLL_AND_FUSE──────────────────────────────────────────────────><
       │                │ ┌─(─unroll_factor─)─┐│
       └─NOUNROLL_AND_FUSE─┘
```

unroll_factor
>   The *unroll_factor* must be a positive scalar integer constant expression. An *unroll_factor* of 1 disables loop unrolling. If you do not specify an *unroll_factor*, loop unrolling is compiler determined.

Rules

You must specify one of the following compiler options to enable loop unrolling:
- **–O3** or higher optimization level
- **-qhot** compiler option
- **-qsmp** compiler option

Note that if the **-qstrict** option is in effect, no loop unrolling will occur. If you want to enable loop unrolling with the **-qhot** option alone, you must also specify **-qnostrict**.

The **UNROLL_AND_FUSE** directive must immediately precede a **DO** loop.

You must not specify the **UNROLL_AND_FUSE** directive for the innermost **DO** loop.

You must not specify the **UNROLL_AND_FUSE** directive more than once, or combine the directive with **NOUNROLL_AND_FUSE**, **NOUNROLL**, **UNROLL**, or **STREAM_UNROLL** directives for the same **DO** construct.

You must not specify the **UNROLL_AND_FUSE** directive for a **DO WHILE** loop or an infinite **DO** loop.

Examples

Example 1: In the following example, the **UNROLL_AND_FUSE** directive replicates and fuses the body of the loop. This reduces the number of cache misses for Array *B*.

```
      INTEGER, DIMENSION(1000, 1000) :: A, B, C
!IBM* UNROLL_AND_FUSE(2)
      DO I = 1, 1000
         DO J = 1, 1000
            A(J,I) = B(I,J) * C(J,I)
         END DO
      END DO
      END
```

The **DO** loop below shows a possible result of applying the **UNROLL_AND_FUSE** directive.

```
      DO I = 1, 1000, 2
         DO J = 1, 1000
            A(J,I) = B(I,J) * C(J,I)
            A(J,I+1) = B(I+1, J) * C(J, I+1)
         END DO
      END DO
```

Example 2: The following example uses multiple **UNROLL_AND_FUSE** directives:

```
      INTEGER, DIMENSION(1000, 1000) :: A, B, C, D, H
!IBM* UNROLL_AND_FUSE(4)
      DO I = 1, 1000
!IBM* UNROLL_AND_FUSE(2)
         DO J = 1, 1000
            DO k = 1, 1000
               A(J,I) = B(I,J) * C(J,I) + D(J,K)*H(I,K)
            END DO
         END DO
      END DO
      END
```

Related information

- For additional methods of optimizing loops, see the **BLOCK_LOOP**, **LOOPID**, **STREAM UNROLL** and the **UNROLL** directives.

Chapter 13. Hardware-specific directives

This section provides an alphabetical reference to hardware-specific compiler directives. Unless otherwise noted, a directive will function on any supported hardware. This section contains the following categories:

Cache control

CACHE_ZERO

Purpose

The **CACHE_ZERO** directive invokes the machine instruction, data cache block set to zero (dcbz). This instruction sets the data cache block corresponding to the variables you specify to zero. Use this directive with discretion.

Syntax

```
►►─CACHE_ZERO─(─cv_var_list─)──────────────────────────────────►◄
```

cv_var   is a variable associated with the cache block that is set to zero. The variable must be a data object with a determinable storage address. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008`, function name, constant, label, zero-sized string, or an array with vector subscripts.

Examples

In the following example, assume that array *ARRA* has already been loaded into a cache block that you want to set to zero. The data in the cache block is then set to zero.

```
   real(4) :: arrA(2**5)
 ! ....
 !IBM* CACHE_ZERO(arrA(1))               ! set data in cache block to zero
```

DCBF

Purpose

The **DCBF** directive copies a modified cache block to main memory and invalidates the copy in the data cache. If the cache block containing *variable* is in the data cache and is modified, it is copied to main memory.

Syntax

```
 ▶▶──DCBF──(──variable──)──────────────────────────────────────▶◀
```

variable

> any data item that can be passed by reference to a subprogram, except for
> a named constant, zero-length array, or an array section with vector
> subscript.

DCBFL

Purpose

The **DCBFL** directive flushes the cache line at the specified prefetch variable from
the L1 data cache only and ensures that the latest version of the target storage
block is still in L2 but no longer in L1.

Syntax

```
 ▶▶──DCBFL──(──variable──)──────────────────────────────────────▶◀
```

variable

> any data item that can be passed by reference to a subprogram, except for
> a named constant, zero-length array, or an array section with vector
> subscript.

DCBFLP

Purpose

The **DCBFLP** directive flushes the cache line at the address from the primary data
cache of a single processor.

Syntax

```
 ▶▶──DCBFLP──(──variable──)──────────────────────────────────────▶◀
```

variable
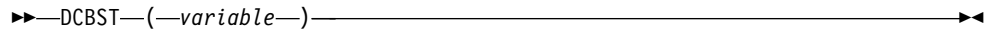
> any data item that can be passed by reference to a subprogram, except for
> a named constant, zero-length array, or an array section with vector
> subscript.

DCBST

Purpose

The **DCBST** directive copies a modified cache block to main memory. If the cache
block containing *variable* is in the data cache and is modified, it is copied to main
memory.

Syntax

```
►►──DCBST──(──variable──)───────────────────────────────────►◄
```

variable

> any data item that can be passed by reference to a subprogram, except for a named constant, zero-length array, or an array section with vector subscript.

EIEIO

Purpose

Enforce In-order Execution of Input/Output (**EIEIO**).

The **EIEIO** directive allows you to specify that all I/O storage access instructions preceding the directive complete before any I/O access instruction subsequent to the directive can begin. Use **EIEIO** when managing shared data instruction where the execution order of load/store access is significant.

EIEIO can provide the necessary functionality for controlling I/O stores without the cost to performance that can occur with other synchronization instructions.

Syntax

```
►►──EIEIO──────────────────────────────────────────────────►◄
```

ISYNC

Purpose

The **ISYNC** directive enables you to discard any prefetched instructions after all preceding instructions complete. Subsequent instructions are fetched or refetched from storage and execute in the context of previous instructions. The directive only affects the processor that executes **ISYNC**.

Syntax

```
►►──ISYNC──────────────────────────────────────────────────►◄
```

LIGHT_SYNC

Purpose

The **LIGHT_SYNC** directive ensures that all stores prior to **LIGHT_SYNC** complete before any new instructions can be executed on the processor that executed the **LIGHT_SYNC** directive. This allows you to synchronize between

multiple processors with minimal performance impact, as **LIGHT_SYNC** does not wait for confirmation from each processor.

Syntax

```
►►──LIGHT_SYNC──────────────────────────────────────────────────►◄
```

PREFETCH

The PREFETCH directive instructs the compiler to load specific data from main memory into the cache before the data is referenced. Some prefetching can be done automatically by hardware, but because compiler-assisted software prefetching can use information directly from your source code, specifying the directive can significantly reduce the number of cache misses.

Rules

When you prefetch a variable, the memory block that includes the variable address is loaded into the cache. A memory block is equal to the size of a cache line. Because the variable you are loading into the cache may appear anywhere within the memory block, you may not be able to prefetch all the elements of an array.

These directives may appear anywhere in your source code where executable constructs may appear.

These directives can add run-time overhead to your program. Therefore you should use the directives only where necessary.

To maximize the effectiveness of the prefetch directives, it is recommended that you specify the **LIGHT_SYNC** directive after a single prefetch or at the end of a series of prefetches.

Related information

For information on applying prefetch techniques to loops with a large iteration count, see the **STREAM_UNROLL** directive.

DCBTSTT
Purpose

The **DCBTSTT** directive fetches the block of memory that contains the specified address into the data cache. Store transient touch provides a hint describing a block that the program might perform a store access to.

Syntax

```
►►──DCBTSTT──(──prefetch_variable──)─────────────────────────────►◄
```

prefetch_variable

> is a variable to be prefetched. The variable must be a data object with a determinable storage address. This variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008` , function name, literal constant, label, zero-sized string, zero-sized array, or array with a vector subscript.

DCBTT

Purpose

The **DCBTT** directive loads the block of memory that contains the specified address into the L1 data cache.

Syntax

```
►►──DCBTT──(──prefetch_variable──)──────────────────────────────►◄
```

prefetch_variable

> is a variable to be prefetched. The variable must be a data object with a determinable storage address. This variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008` , function name, literal constant, label, zero-sized string, zero-sized array, or array with a vector subscript.

DEFAULT_PREFETCH_DEPTH

Purpose

The **DEFAULT_PREFETCH_DEPTH** directive defines a prefetch depth for hardware-detected and software-defined streams. This directive takes effect in one of the following cases:

- When 0 is specified for the prefetch depth
- When __dcbt or __dcbtst with TH=1010 is not used

Syntax

```
►►──DEFAULT_PREFETCH_DEPTH──(──prefetch_depth──)──────────────────►◄
```

prefetch_depth

> Must be a scalar and of type integer with a value from 0 to 7. It is a relative value with the defined levels as follows:

Table 57. Urgency

Integer value	Urgency
0	default
1	not urgent
2	the least urgent
3	less urgent
4	medium
5	urgent
6	more urgent
7	the most urgent

DEPTH_ATTAINMENT_URGENCY

Purpose

The **DEPTH_ATTAINMENT_URGENCY** directive sets the urgency of reaching the prefetch depth for hardware-detected streams.

Syntax

```
►►──DEPTH_ATTAINMENT_URGENCY──(──urgency──)───────────────────────────────►◄
```

urgency

> Must be a scalar and of type integer with a value from 0 to 7. It is a relative value with the defined levels as follows:

Table 58. Urgency

Integer value	Urgency
0	Default
1	Not urgent
2	The least urgent
3	Less urgent
4	Medium
5	Urgent
6	More urgent
7	The most urgent

HARDWARE_TRANSIENT_ENABLE

Purpose

The **HARDWARE_TRANSIENT_ENABLE** directive enables or disables the transient attribute for hardware-detected streams.

Syntax

```
►►──HARDWARE_TRANSIENT_ENABLE──(──flag──)────────────────────────────────►◄
```

Note: If you want the Data Stream Control Register (DSCR) to apply the transient attribute to streams, set at least two of the four enabling built-in functions. Set one function to choose a type of access (load or store), and set another one to choose a type of prefetching (software-defined or hardware-detected).

flag   A scalar logical expression. The value .FALSE. disables the transient attribute for hardware-detected streams. The value .TRUE. enables the transient attribute for hardware-detected streams.

HARDWARE_UNIT_COUNT_ENABLE
Purpose

The **HARDWARE_UNIT_COUNT_ENABLE** directive enables or disables the unit count for hardware-detected streams. The unit count is a scalar and of type integer with a value from 0 to 1023, which indicates the number of cache lines.

Syntax

```
►►──HARDWARE_UNIT_COUNT_ENABLE──(──flag──)───────────────────────────────►◄
```

flag   A scalar logical expression. The value .TRUE. enables the unit count for hardware-detected streams. The value .FALSE. disables the unit count for hardware-detected streams.

LOAD_STREAM_DISABLE
Purpose

The **LOAD_STREAM_DISABLE** directive disables hardware detection and the initiation of loading streams.

Syntax

```
►►──LOAD_STREAM_DISABLE──(──flag──)──────────────────────────────────────►◄
```

flag   A scalar logical expression. The value .FALSE. disables the hardware detection and the initiation of loading streams. The value .TRUE. enables the hardware detection and the initiation of loading streams.

LOAD_TRANSIENT_ENABLE

Purpose

The **LOAD_TRANSIENT_ENABLE** directive enables or disables the transient attribute for load streams.

Syntax

```
►►──LOAD_TRANSIENT_ENABLE──(──flag──)──────────────────────────────►◄
```

Note: If you want the Data Stream Control Register (DSCR) to apply the transient attribute to streams, set at least two of the four enabling built-in functions. Set one function to choose a type of access (load or store), and set another one to choose a type of prefetching (software-defined or hardware-detected).

flag   A scalar logical expression. The value .TRUE. enables the transient attribute for load streams. The value .FALSE. disables the transient attribute for load streams.

PARTIAL_DCBT

Purpose

The **PARTIAL_DCBT** directive loads half of the cache line that contains the specified address into the L3 data cache.

Syntax

```
►►──PARTIAL_DCBT──(──prefetch_variable──)──────────────────────────►◄
```

prefetch_variable
>is a variable to be prefetched. The variable must be a data object with a determinable storage address. This variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008`, function name, literal constant, label, zero-sized string, zero-sized array, or array with a vector subscript.

PREFETCH_BY_LOAD

Purpose

The **PREFETCH_BY_LOAD** directive prefetches data into the cache by way of a load instruction. **PREFETCH_BY_LOAD** enables hardware-assisted prefetching.

Syntax

```
  ►►──PREFETCH_BY_LOAD──(──prefetch_variable_list──)─────────────────────────►◄
```

prefetch_variable

> is a variable to be prefetched. The variable must be a data object with a
> determinable storage address. The variable can be of any data type,
> including intrinsic and derived data types. The variable cannot be a
> procedure name, subroutine name, module name, `F2008` submodule name
> `F2008`, function name, constant, label, zero-sized string, or an array with a
> vector subscript.

PREFETCH_BY_STREAM
Purpose

The **PREFETCH_BY_STREAM** prefetch technique uses the processor to recognize
sequential access to adjacent cache lines and then requests anticipated lines from
deeper levels of the memory hierarchy. When repeated references to main memory
are made, this technique establishes a path or stream, which increases the depth of
the prefetch until enough lines are loaded into the cache. To fetch data from
decremental memory addresses, use the **PREFETCH_BY_STREAM_BACKWARD**
directive. To fetch data from incremental memory addresses, use the
PREFETCH_BY_STREAM_FORWARD directive. The use of this streamed prefetch
to load data from main memory into the cache can reduce or eliminate load
latency.

Syntax

```
  ►►──PREFETCH_BY_STREAM_BACKWARD──(──prefetch_variable──)──────────────────►◄
```

```
  ►►──PREFETCH_BY_STREAM_FORWARD──(──prefetch_variable──)───────────────────►◄
```

prefetch_variable

> is a variable to be prefetched. The variable must be a data object with a
> determinable storage address. The variable can be of any data type,
> including intrinsic and derived data types. The variable cannot be a
> procedure name, subroutine name, module name, `F2008` submodule name
> `F2008`, function name, constant, label, zero-sized string, or an array with a
> vector subscript.

PREFETCH_FOR_LOAD
Purpose

The **PREFETCH_FOR_LOAD** directive prefetches data into the cache for reading
by way of a cache prefetch instruction.

Syntax

```
►►──PREFETCH_FOR_LOAD──(──prefetch_variable_list──)──────────────────────►◄
```

prefetch_variable
> is a variable to be prefetched. The variable must be a data object with a
> determinable storage address. The variable can be of any data type,
> including intrinsic and derived data types. The variable cannot be a
> procedure name, subroutine name, module name, `F2008` submodule name
> `F2008` , function name, constant, label, zero-sized string, or an array with a
> vector subscript.

PREFETCH_FOR_STORE
Purpose

The **PREFETCH_FOR_STORE** directive prefetches data into the cache for writing
by way of a cache prefetch instruction.

Syntax

```
►►──PREFETCH_FOR_STORE──(──prefetch_variable_list──)──────────────────────►◄
```

prefetch_variable
> is a variable to be prefetched. The variable must be a data object with a
> determinable storage address. The variable can be of any data type,
> including intrinsic and derived data types. The variable cannot be a
> procedure name, subroutine name, module name, `F2008` submodule name
> `F2008` , function name, constant, label, zero-sized string, or an array with a
> vector subscript.

PREFETCH_GET_DSCR_REGISTER
Purpose

The **PREFETCH_GET_DSCR_REGISTER** intrinsic procedure returns the current
value of the Data Stream Control Register (DSCR). The DSCR is made accessible
for controlling data stream prefetch and the speed of the ramp.

Class

Function

Argument type and attributes

None

Result type and attributes

The result is of type `INTEGER (8)`.

PREFETCH_SET_DSCR_REGISTER(VALUE)

Purpose

The **PREFETCH_SET_DSCR_REGISTER** intrinsic procedure sets the Data Stream Control Register (DSCR) value to the input value. The DSCR is made accessible for controlling data stream prefetch and the speed of the ramp.

Class

Subroutine

Argument type and attributes

VALUE
An `INTENT(IN) INTEGER (8)`.

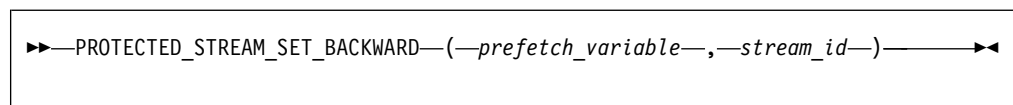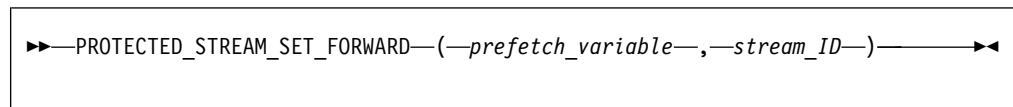## Result type and attributes

None

PROTECTED_STORE_STREAM_SET

Purpose

The **PROTECTED_STORE_STREAM_SET_FORWARD** directive establishes a limited-length protected store stream that begins with the cache line at the specified prefetch variable and fetches from increasing memory addresses. The **PROTECTED_STORE_STREAM_SET_BACKWARD** directive fetches from decreasing memory addresses.

Syntax

```
►►──PROTECTED_STORE_STREAM_SET_FORWARD──(──prefetch_variable──,──stream_ID──)────────►◄
```

```
►►──PROTECTED_STORE_STREAM_SET_BACKWARD──(──prefetch_variable──,──stream_id──)───────►◄
```

prefetch_variable
is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, F2008 submodule name F2008 , function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

stream_id
is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8® or higher processors, it can be any number from 0 to 11.

PROTECTED_STREAM_COUNT

Purpose

The **PROTECTED_STREAM_COUNT** directive sets the number of cache lines for the specified limited-length stream.

Syntax

```
►►──PROTECTED_STREAM_COUNT──(──unit_count──,──stream_id──)────────────►◄
```

unit_count

> is the number of cache lines for the limited-length protected stream. It must be scalar and of type integer. It can be any number from 0 to 1023. For a stream that is larger than 1024 cache lines, use the **PROTECTED_UNLIMITED_STREAM** directives instead of the **PROTECTED_STREAM** directives.

stream_id

> is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_STREAM_COUNT_DEPTH

Purpose

The **PROTECTED_STREAM_COUNT_DEPTH** directive sets the number of cache lines and the prefetch depth for the specified limited-length stream.

Syntax

```
►►──PROTECTED_STREAM_COUNT_DEPTH──(──unit_count──,──prefetch_depth──,──stream_id──)────────►◄
```

unit_count

> is the number of cache lines for the limited-length protected stream. It must be scalar and of type integer. It can be any number from 0 to 1023. For a stream that is larger than 1024 cache lines, use the **PROTECTED_UNLIMITED_STREAM** directives instead of the **PROTECTED_STREAM** directives.

prefetch_depth

> can be specified by either an integer variable or a numeric literal ranging from 0 to 7. It is a relative value with the levels defined qualitatively as follows:

Table 59. Prefetch depth

Integer value	Depth
0	default
1	none

Table 59. Prefetch depth (continued)

Integer value	Depth
2	shallowest
3	shallow
4	medium
5	deep
6	deeper
7	deepest

stream_id
>   is the ID for the prefetched stream. It must be scalar and of type integer.
>   For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_STREAM_GO

Purpose

The **PROTECTED_STREAM_GO** directive starts to prefetch all limited-length
streams.

Syntax

```
►►──PROTECTED_STREAM_GO────────────────────────────────────────►◄
```

PROTECTED_STREAM_SET

Purpose

The **PROTECTED_STREAM_SET_FORWARD** directive establishes a
limited-length protected stream that begins with the cache line at the specified
prefetch variable and fetches from increasing memory addresses. The
PROTECTED_STREAM_SET_BACKWARD directive fetches from decreasing
memory addresses.

Syntax

```
►►──PROTECTED_STREAM_SET_FORWARD──(──prefetch_variable──,──stream_ID──)────────►◄
```

```
►►──PROTECTED_STREAM_SET_BACKWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

prefetch_variable
>   is a variable to be prefetched. The variable must be a data object with a
>   determinable storage address. The variable can be of any data type,
>   including intrinsic and derived data types. The variable cannot be a

procedure name, subroutine name, module name, `F2008` submodule name `F2008` , function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

stream_id
>   is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_STREAM_STRIDE

Purpose

The **PROTECTED_STREAM_STRIDE** directive sets the distance between each element of the specified prefetch stream.

Syntax

```
►►──PROTECTED_STREAM_STRIDE──(──address_offset,──stride,──stream_id──)────────►◄
```

address_offset
>   is the word-offset of the first unit of the stream to be prefetched, that is, the number of words away from the start of a cache line that the address of the first unit is. This variable must be scalar and of type integer. Here is an example of how to acquire address_offset:
>
>   i8 = LOC(*variable_name*)
>
>   *address_offset* = ISHFT(IAND(i8,127_8),-2)
>
>   i8 is of type INTEGER(8) and *address_offset* is of type INTEGER(4) and *variable_name* can be of any type.

stride   is the distance in bytes between each element of the prefetch stream in word size. This variable must be scalar and of type integer and should be in the range of 0 to 262143.

stream_id
>   is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_STREAM_STOP

Purpose

The **PROTECTED_STREAM_STOP** directive stops prefetching the specified protected stream.

Syntax

```
►►──PROTECTED_STREAM_STOP──(──stream_id──)──────────────────────────────────►◄
```

stream_id
> is the ID for the prefetched stream. It must be scalar and of type integer.

PROTECTED_STREAM_STOP_ALL
Purpose

The **PROTECTED_STREAM_STOP_ALL** directive stops prefetching all protected streams.

Syntax

```
►►──PROTECTED_STREAM_STOP_ALL────────────────────────────────────────►◄
```

PROTECTED_UNLIMITED_STORE_STREAM _SET
Purpose

The **PROTECTED_UNLIMITED_STORE_STREAM_SET_FORWARD** directive establishes an unlimited-length protected store stream that begins with the cache line at the specified prefetch variable and fetches from increasing memory addresses. The **PROTECTED_UNLIMITED_STORE_STREAM_SET_BACKWARD** directive fetches from decreasing memory addresses.

Syntax

```
►►──PROTECTED_UNLIMITED_STORE_STREAM_SET_FORWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

```
►►──PROTECTED_UNLIMITED_STORE_STREAM_SET_BACKWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

prefetch_variable
> is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008`, function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

stream_id
> is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_UNLIMITED_STREAM_SET
Purpose

The **PROTECTED_UNLIMITED_STREAM_SET_FORWARD** directive establishes an unlimited-length protected stream that begins with the cache line at the specified prefetch variable and fetches from increasing memory addresses. The

PROTECTED_UNLIMITED_STREAM_SET_BACKWARD directive fetches from decreasing memory addresses.

Syntax

```
►►──PROTECTED_UNLIMITED_STREAM_SET_FORWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

```
►►──PROTECTED_UNLIMITED_STREAM_SET_BACKWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

prefetch_variable
> is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008`, function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

stream_id
> is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

PROTECTED_UNLIMITED_STREAM_SET_GO
Purpose

The **PROTECTED_UNLIMITED_STREAM_SET_GO_FORWARD** directive establishes an unlimited-length protected stream that begins with the cache line at the specified prefetch variable. This directive fetches the data from increasing memory addresses. The **PROTECTED_UNLIMITED_STREAM_SET_GO_BACKWARD** directive fetches from decreasing memory addresses.

Syntax

```
►►──PROTECTED_UNLIMITED_STREAM_SET_GO_FORWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

```
►►──PROTECTED_UNLIMITED_STREAM_SET_GO_BACKWARD──(──prefetch_variable──,──stream_id──)────────►◄
```

prefetch_variable
> is a variable to be prefetched. The variable must be a data object with a determinable storage address. The variable can be of any data type, including intrinsic and derived data types. The variable cannot be a procedure name, subroutine name, module name, `F2008` submodule name `F2008`, function name, literal constant, label, zero-sized string, zero-length array, or array with a vector subscript.

stream_id
> is the ID for the prefetched stream. It must be scalar and of type integer. Its value can be any number from 0 to 11.

SET_PREFETCH_UNIT_COUNT
Purpose

The **SET_PREFETCH_UNIT_COUNT** directive sets the number of units in data stream.

Syntax

```
►►──SET_PREFETCH_UNIT_COUNT──(──count──)────────────────────────►◄
```

count The number of cache lines. Must be a scalar of type integer with a value from 0 to 1023.

SOFTWARE_TRANSIENT_ENABLE
Purpose

The **SOFTWARE_TRANSIENT_ENABLE** directive enables or disables the transient attribute for software-defined streams.

Syntax

```
►►──SOFTWARE_TRANSIENT_ENABLE──(──flag──)───────────────────────►◄
```

Note: If you want the Data Stream Control Register (DSCR) to apply the transient attribute to streams, set at least two of the four enabling built-in functions. Set one function to choose a type of access (load or store), and set another one to choose a type of prefetching (software-defined or hardware-detected).

flag A scalar logical expression. The value .TRUE. enables the transient attribute for software-defined streams. The value .FALSE. disables the transient attribute for software-defined streams.

SOFTWARE_UNIT_COUNT_ENABLE
Purpose

The **SOFTWARE_UNIT_COUNT_ENABLE** directive enables or disables the unit count for software-defined streams. The unit count is a scalar and of type integer. It has a value from 0 to 1023, which indicates the number of cache lines.

Syntax

```
►►──SOFTWARE_UNIT_COUNT_ENABLE──(──flag──)────────────────────────────────►◄
```

flag   A scalar logical expression. The value .TRUE. enables the unit count for
           software-defined streams. The value .FALSE. disables the unit count for
           software-defined streams.

STORE_TRANSIENT_ENABLE
Purpose

The **STORE_TRANSIENT_ENABLE** directive enables or disables the transient
attribute for store streams.

Syntax

```
►►──STORE_TRANSIENT_ENABLE──(──flag──)────────────────────────────────────►◄
```

Note: If you want the Data Stream Control Register (DSCR) to apply the transient
attribute to streams, set at least two of the four enabling built-in functions. Set one
function to choose a type of access (load or store), and set another one to choose a
type of prefetching (software-defined or hardware-detected).

flag   A scalar logical expression. The value .TRUE. enables the transient attribute
           for store streams. The value .FALSE. disables the transient attribute for
           store streams.

STRIDE_N_STREAM_ENABLE
Purpose

The **STRIDE_N_STREAM_ENABLE** directive enables the hardware detection and
initiation of loading and storing streams that have a stride greater than a single
cache block. Such loading streams are detected only when the value of Load
Stream Disable (LSD) bit is 0. Such storing streams are detected only when the
value of Store Stream Enable (SSE) bit is 1.

Syntax

```
►►──STRIDE_N_STREAM_ENABLE──(──flag──)────────────────────────────────────►◄
```

flag   A scalar logical expression. The value .TRUE. enables the hardware
           detection and initiation of loading and storing streams that have a stride
           greater than a single cache block. The value .FALSE. disables it.

TRANSIENT_PROTECTED_STREAM_COUNT _DEPTH

Purpose

The **TRANSIENT_PROTECTED_STREAM_COUNT_DEPTH** directive sets the number of cache lines and the prefetch depth for the specified limited-length prefetch stream.

Syntax

```
►►──TRANSIENT_PROTECTED_STREAM_COUNT_DEPTH──(──unit_count,──prefetch_depth,──stream_id──)──────────►◄
```

unit_count

        is the number of cache lines to be prefetched. This must be scalar and of type integer and should be in the range of 0 to 1023.

prefetch_depth

        sets the depth of prefetching as described above. This must be scalar and of type integer and should be in the range of 0 to 7.

stream_id

        is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

TRANSIENT_UNLIMITED_PROTECTED _STREAM_DEPTH

Purpose

The **TRANSIENT_UNLIMITED_PROTECTED_STREAM_DEPTH** directive sets the prefetch depth for the specified unlimited-length prefetch stream.

Syntax

```
►►──TRANSIENT_UNLIMITED_PROTECTED_STREAM_DEPTH──(──prefetch_depth,──stream_id──)────────►◄
```

prefetch_depth

        sets the depth of prefetching as described above. This must be scalar and of type integer and should be in the range of 0 to 7.

stream_id

        is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

UNLIMITED_PROTECTED_STREAM_DEPTH

Purpose

The **UNLIMITED_PROTECTED_STREAM_DEPTH** directive sets the prefetch depth for the specified unlimited-length prefetch stream.

Syntax

```
►►──UNLIMITED_PROTECTED_STREAM_DEPTH──(──unit_count,──prefetch_depth,──stream_id──)────────►◄
```

unit_count
> is the number of cache lines to be prefetched. This must be scalar and of type integer and should be in the range of 0 to 1023.

prefetch_depth
> sets the depth of prefetching as described above. This must be scalar and of type integer and should be in the range of 0 to 7.

stream_id
> is the ID for the prefetched stream. It must be scalar and of type integer. For POWER8 or higher processors, it can be any number from 0 to 11.

Examples
Examples

Example 1: This example shows valid uses of the **PREFETCH_BY_LOAD**, **PREFETCH_FOR_LOAD**, and **PREFETCH_FOR_STORE** directives.

For this example, assume that the size of the cache line is 64 bytes and that none of the declared data items exist in the cache at the beginning of the program. The rationale for using the directives is as follows:

- All elements of array *ARRA* will be assigned; therefore, you can use the **PREFETCH_FOR_STORE** directive to bring the first 16 and second 16 elements of the array into the cache before they are referenced.

- Because all elements of array *ARRC* will be read, you can use the **PREFETCH_FOR_LOAD** directive to bring the first 16 and second 16 elements of the array into the cache before they are referenced. (Assume that the elements have been initialized first.)

- Each iteration of the loop will use variables *A*, *B*, *C*, *TEMP*, *I*, *K* and array element *ARRB(I*32)*; you can use the **PREFETCH_BY_LOAD** directive to load the variables and the array into the cache. (Because of the size of the cache line, you will fetch 16 elements of *ARRB*, starting at element *ARRB(I*32)*).

```
    PROGRAM GOODPREFETCH

    REAL*4 A, B, C, TEMP
    REAL*4 ARRA(2**5), ARRB(2**10), ARRC(2**5)
    INTEGER(4) I, K

! Bring ARRA into cache for writing.
!IBM* PREFETCH_FOR_STORE (ARRA(1), ARRA(2**4+1))

! Bring ARRC into cache for reading.
!IBM* PREFETCH_FOR_LOAD (ARRC(1), ARRC(2**4+1))

! Bring all variables into the cache.
!IBM* PREFETCH_BY_LOAD (A, B, C, TEMP, I , K)

! A subroutine is called to allow clock cycles to pass so that the
! data is loaded into the cache before the data is referenced.
    CALL FOO()
    K = 32
    DO I = 1, 2 ** 5

! Bring ARRB(I*K) into the cache
!IBM* PREFETCH_BY_LOAD (ARRB(I*K))
```

```
         A = -I
         B = I + 1
         C = I + 2
         TEMP = SQRT(B*B - 4*A*C)
         ARRA(I) = ARRC(I) + (-B + TEMP) / (2*A)
         ARRB(I*K) = (-B - TEMP) / (2*A)
       END DO
       END PROGRAM GOODPREFETCH
```

Example 2: In this example, assume that the total cache line's size is 256 bytes, and that none of the declared data items are initially stored in the cache or register. All elements of array *ARRA* and *ARRC* will then be read into the cache.

```
     PROGRAM PREFETCH_STREAM

     REAL*4 A, B, C, TEMP
     REAL*4 ARRA(2**5), ARRC(2**5), ARRB(2**10)
     INTEGER*4 I, K

! All elements of ARRA and ARRC are read into the cache.
!IBM* PREFETCH_BY_STREAM_FORWARD(ARRA(1))
! You can substitute PREFETCH_BY_STREAM_BACKWARD (ARRC(2**5)) to read all
! elements of ARRA and ARRC into the cache.
     K = 32
     DO I = 1, 2**5
        A = -i
        B = i + 1
        C = i + 2
        TEMP = SQRT(B*B -4*A*C)
        ARRA(I) = ARRC(I) + (-B + TEMP) / (2*A)
        ARRB(I*K) = (-B -TEMP) / (2*A)
     END DO
     END PROGRAM PREFETCH_STREAM
```

Chapter 14. Intrinsic procedures

Fortran defines a number of procedures, called intrinsic procedures, that are available to any program. This section provides an alphabetical reference to these procedures.

Classes of intrinsic procedures

There are five classes of intrinsic procedures: inquiry functions, elemental procedures, system inquiry functions, transformational functions, and subroutines.

Inquiry intrinsic functions

The result of an *inquiry function* depends on the properties of its principal argument, not on the value of the argument. The value of the argument does not have to be defined.

ALLOCATED	LEN	RANK **4**
ASSOCIATED	LOC **3**	SAME_TYPE_AS **1**
BIT_SIZE	MAXEXPONENT	SHAPE
COMMAND_ARGUMENT_COUNT **1**	MINEXPONENT	SIZE
DIGITS	NEW_LINE **1**	SIZEOF **3**
EPSILON	NUM_PARTHDS **3**	STORAGE_SIZE **2**
EXTENDS_TYPE_OF **1**	NUM_USRTHDS **3**	TINY
IS_CONTIGUOUS **2**	PRECISION	UBOUND
HUGE	PRESENT	
KIND	RADIX	
LBOUND	RANGE	
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension **4** TS 29113		

Elemental intrinsic procedures

Some intrinsic functions and one intrinsic subroutine (MVBITS) are *elemental*. That is, they can be specified for scalar arguments, but also accept arguments that are arrays.

If all arguments are scalar, the result is a scalar.

If any argument is an array, all INTENT(OUT) and INTENT(INOUT) arguments must be arrays of the same shape, and the remaining arguments must be conformable with them.

The shape of the result is the shape of the argument with the greatest rank. The elements of the result are the same as if the function was applied individually to the corresponding elements of each argument.

ABS [3]	FLOOR	MAX
ACHAR	FRACTION	MERGE
ACOS	GAMMA [2]	MERGE_BITS [2]
ACOSD	HFIX [2]	MIN
ACOSH [2]	HYPOT [3]	MOD
ADJUSTL	IACHAR	MODULO
ADJUSTR	IAND	MVBITS
AIMAG	IBCLR	NEAREST
AINT	IBM2GCCLDBL [3]	NINT
ASIN	IBM2GCCLDBL_CMPLX [3]	NOT
ASIND [3]	IBSET	POPCNT [2]
ASINH [2]	ICHAR	POPCNTB
ATAN	IEOR	POPPAR [1]
ATAND [3]	ILEN [3]	QCMPLX [3]
ATAN2	INDEX	QEXT [3]
ATAN2D [3]	INT	REAL
BTEST	INT2 [3]	RRSPACING
CEILING	IOR	RSHIFT
CHAR	ISHFT	SCALE
CMPLX	ISHFTC	SCAN
CONJG	IS_IOSTAT_END [1]	SET_EXPONENT
COS	IS_IOSTAT_EOR [2]	SHIFTA [2]
COSD [3]	LEADZ [2]	SHIFTL [2]
COSH	LEN_TRIM	SHIFTR [2]
CVMGx [3]	LGAMMA [3]	SIGN
DBLE	LGE	SIN
DCMPLX [3]	LGT	SIND [3]
DIM	LLE	SINH
DPROD	LLT	SPACING
DSHIFTL [2]	LOG	SQRT
DSHIFTR [2]	LOG_GAMMA [2]	TAN
ERF [2]	LOG10	TAND [3]
ERFC [2]	LOGICAL	TANH
ERFC_SCALED [2]	LSHIFT [3]	TRAILZ [2]
EXP	MASKL [2]	VERIFY
EXPONENT	MASKR [2]	
Note: [1] Fortran 2003 [2] Fortran 2008 [3] IBM extension		

System inquiry intrinsic functions (IBM extension)

The *system inquiry functions* may be used in specification expressions. They cannot be used in constant expressions, nor can they be passed as actual arguments.

- NUMBER_OF_PROCESSORS
- PROCESSORS_SHAPE

Transformational intrinsic functions

All other intrinsic functions are classified as *transformational functions*. They generally accept array arguments and return array results that depend on the values of elements in the argument arrays.

ALL	MAXVAL	SELECTED_INT_KIND
ANY	MINLOC	SELECTED_REAL_KIND
COUNT	MINVAL	SPREAD
CSHIFT	NULL	SUM
DOT_PRODUCT	PACK	TRANSFER
EOSHIFT	PRODUCT	TRANSPOSE
FINDLOC **1**	REPEAT	TRIM
MATMUL	RESHAPE	UNPACK
MAXLOC	SELECTED_CHAR_KIND **2**	
Note: **1** Fortran 2008 **2** Fortran 2003		

For background information on arrays, see Chapter 5, "Array concepts," on page 75.

Intrinsic subroutines

Some intrinsic procedures are subroutines. They perform various tasks.

ALIGNX **3**	MOVE_ALLOC **1**
ABORT **3**	MVBITS
CPU_TIME	RANDOM_NUMBER
DATE_AND_TIME	RANDOM_SEED
EXECUTE_COMMAND_LINE **2**	SIGNAL **3**
GETENV	SRAND **3**
GET_COMMAND **1**	SYSTEM **3**
GET_COMMAND_ARGUMENT **1**	SYSTEM_CLOCK
GET_ENVIRONMENT_VARIABLE **1**	
Note: **1** Fortran 2003 **2** Fortran 2008 **3** IBM extension	

Data representation models

Integer bit model

The following model shows how the processor represents each bit of a nonnegative scalar integer object:

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

j is the integer value

s is the number of bits

w_k is binary digit w located at position k

--- IBM extension ---

XL Fortran implements the following s parameters for the XL Fortran integer kind type parameters:

Integer Kind Parameter	s Parameter
1	8
2	16
4	32
8	64

--- End of IBM extension ---

The following intrinsic functions use this model:

BTEST	IBSET	ISHFTC
IAND	IEOR	MVBITS
IBCLR	IOR	NOT
IBITS	ISHFT	

Integer data model

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

i is the integer value

s is the sign (± 1)

q is the number of digits (positive integer)

w_k is a nonnegative digit $< r$

	r is the radix

XL Fortran implements this model with the following r and q parameters:

Integer Kind Parameter	r Parameter	q Parameter
1	2	7
2	2	15
4	2	31
8	2	63

The following intrinsic functions use this model:
- DIGITS
- HUGE
- RADIX
- RANGE

Real data model

$$x = \begin{cases} 0 & \text{or} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \end{cases}$$

x is the real value

s is the sign (± 1)

b is an integer > 1

e is an integer, where $e_{min} \leq e \leq e_{max}$

p is an integer > 1

f_k is a nonnegative integer $< b$ ($f_1 \neq 0$)

Note: If x=0, then e=0 and all f_k=0.

XL Fortran implements this model with the following parameters:

Real Kind parameter	b Parameter	p Parameter	e_{min} Parameter	e_{max} Parameter
4	2	24	-125	128
8	2	53	-1021	1024
16	2	106	-1021	1024

The following intrinsic functions use this model:

DIGITS	MINEXPONENT	RRSPACING
EPSILON	NEAREST	SCALE
EXPONENT	PRECISION	SET_EXPONENT
FRACTION	RADIX	SPACING
HUGE	RANGE	TINY
MAXEXPONENT		

Detailed descriptions of intrinsic procedures

The following is an alphabetical list of all generic names for intrinsic procedures.

For each procedure, several items of information are listed.

Note:

1. The argument names listed in the title can be used as the names for keyword arguments when calling the procedure.
2. For those procedures with specific names, a table lists each specific name along with information about the specific function:
   - When a function return type or argument type is shown in lowercase, that indicates that the type is specified as shown, but the compiler may actually substitute a call to a different specific name depending on the settings of the **-qintsize**, **-qrealsize**, and **-qautodbl** options.

     For example, references to **SINH** are replaced by references to **DSINH** when **-qrealsize=8** is in effect, and references to **DSINH** are replaced by references to **QSINH**.
   - The column labeled "Pass as Arg?" indicates whether or not you can pass that specific name as an actual argument to a procedure. Only the specific name of an intrinsic procedure may be passed as an actual argument, and only for some specific names. A specific name passed this way may only be referenced with scalar arguments.
3. The index contains entries for each specific name, if you know the specific name but not the generic one.

ABORT() (IBM extension)
Purpose

Terminates the program abnormally, unless the signal **SIGABRT** is being caught and the signal handler does not return. It truncates all open output files to the current position of the file pointer, closes all open files, and then calls the **abort()** system routine. This results in a **SIGABRT** signal sent to the current process.

The **ABORT** intrinsic overrides blocking or ignoring the **SIGABRT** signal; it will not return.

Class

Subroutine

Examples

The following is an example of a statement using the **ABORT** subroutine.

```
IF (ERROR_CONDITION) CALL ABORT
```

ABS(A)

Purpose

Absolute value.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER, REAL, or COMPLEX

Result type and attributes

The same as A, except that if A is complex, the result is real.

Result value

- If A is of type integer or real, the result is |A|
- If A is of type complex with value (x,y), the result approximates

$$\sqrt{x^2 + y^2}$$

Examples

ABS ((3.0, 4.0)) has the value 5.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
IABS	any integer **1** **2**	same as argument	yes
ABS	default real	default real	yes
DABS	double precision real	double precision real	yes
QABS **1**	REAL(16)	REAL(16)	yes
CABS	default complex	default real	yes
CDABS **1**	double complex	double precision real	yes
ZABS **1**	double complex	double precision real	yes
CQABS **1**	COMPLEX(16)	REAL(16)	yes
Note: **1** IBM extension **2** the ability to specify a nondefault integer argument.			

ACHAR(I, KIND)

Purpose

Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

`F2003` **KIND (optional)**
An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression. `F2003`

Result type and attributes

- Character of length one.
- `F2003` If **KIND** is present, the kind type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default character type. `F2003`

Result value

- If I has a value in the range $0 \le I \le 127$, the result is the character in position I of the ASCII collating sequence, provided that the character corresponding to I is representable.
- If I is outside the allowed value range, the result is undefined.

Examples

ACHAR (88) has the value 'X'.

ACOS(X)

Purpose

Arccosine (inverse cosine) function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL `F2008` or COMPLEX. `F2008` If it's a REAL, its value must satisfy the inequality $|X| \le 1$.

Result type and attributes

Same as X.

Result value

If X is of type real, the result value is as follows:
- It is expressed in radians, and approximates arccos(X).
- It is in the range $0 \leq ACOS(X) \leq \pi$.

F2008

If X is of type complex, the real part of the result value is as follows:
- It is expressed in radians.
- It is in the range $0 \leq REAL(ACOS(X)) \leq \pi$.

F2008

Examples

ACOS(1.0) has the value 0.0.

F2008 **ACOS**((0.540302, 0.000000)) has the value (1.000000, 0.000000), approximately. F2008

Specific Name	Argument Type	Result Type	Pass As Arg?
ACOS	default real	default real	yes
DACOS	double precision real	double precision real	yes
QACOS **1**	REAL(16)	REAL(16)	yes
QARCOS **1**	REAL(16)	REAL(16)	yes

Note:
1. IBM extension.

ACOSD(X) (IBM extension)
Purpose

Arccosine (inverse cosine) function. Result in degrees.

Class

Elemental function

Argument type and attributes

X An `INTENT(IN)` REAL. Its value must satisfy the inequality $|X| \leq 1$.

Result type and attributes

Same as X.

Result value
- It is expressed in degrees and approximates arccos(X).
- It is in the range $0° \leq ACOSD(X) \leq 180°$.

Examples

ACOSD (0.5) has the value 60.0°.

Specific Name	Argument Type	Result Type	Pass As Arg?
ACOSD	default real	default real	yes
DACOSD	double precision real	double precision real	yes
QACOSD	REAL(16)	REAL(16)	yes

ACOSH(X) (Fortran 2008)
Purpose

Inverse hyperbolic cosine function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL or COMPLEX

Result type and attributes

Same as X.

Result value

The result value approximates the inverse hyperbolic cosine of X.

If X is of type complex, the imaginary part of the result value is as follows:
- It is expressed in radians.
- It is in the range $0 \leq AIMAG(ACOSH(X)) \leq \pi$.

Examples

ACOSH(1.5430806) has the value 1.0, approximately.

ACOSH((1.5430806, 0.000000)) has the value (1.000000, 0.000000), approximately.

ADJUSTL(STRING)
Purpose

Adjust to the left, removing leading blanks and inserting trailing blanks.

Class

Elemental function

Argument type and attributes

STRING
     An INTENT(IN) CHARACTER

Result type and attributes

Character of the same length and kind type parameter as STRING.

Result value

The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Examples

ADJUSTL (' WORD') has the value 'WORD '.

ADJUSTR(STRING)
Purpose

Adjust to the right, removing trailing blanks and inserting leading blanks.

Class

Elemental function

Argument type and attributes

STRING
      An `INTENT(IN)` `CHARACTER`

Result type and attributes

Character of the same length and kind type parameter as STRING.

Result value

The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Examples

ADJUSTR ('WORD ') has the value ' WORD'.

AIMAG(Z), IMAG(Z)
Purpose

Imaginary part of a complex number.

Class

Elemental function

Argument type and attributes

Z   An `INTENT(IN)` `COMPLEX`

Result type and attributes

Real with the same kind type parameter as Z.

Result value

If Z has the value (x,y), the result has the value y.

Examples

AIMAG ((2.0, 3.0)) has the value 3.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
AIMAG	default complex	default real	yes
DIMAG **1**	double complex	double precision real	yes
QIMAG **1**	COMPLEX(16)	REAL(16)	yes

Note:

1. IBM extension.

`F2008` In Fortran 2008, you can use *designator*%IM to access the imaginary part of complex numbers directly; for instance, Z%IM has the same value as AIMAG(Z). For more information about complex part designators, see Complex. `F2008`

AINT(A, KIND)
Purpose

Truncates to a whole number.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN)` REAL

KIND (optional)
       An `INTENT(IN)` scalar `INTEGER`. The actual argument corresponding to
       KIND must be a constant expression.

Result type and attributes

- The result type is real.
- If **KIND** is present, the kind type parameter is that specified by **KIND**;
  otherwise, the kind type parameter is that of A.

Result value

- If $|A| < 1$, the result is zero.
- If $|A| \geq 1$, the result has a value equal to the integer whose magnitude is the
  largest integer that does not exceed the magnitude of A and whose sign is the
  same as the sign of A.

Examples

```
AINT(3.555) = 3.0
AINT(-3.555) = -3.0
```

Specific Name	Argument Type	Result Type	Pass As Arg?
AINT	default real	default real	yes
DINT	double precision real	double precision real	yes
QINT **1**	REAL(16)	REAL(16)	yes

Note:

1. IBM extension.

ALIGNX(K,M) (IBM extension)
Purpose

The **ALIGNX** built-in subroutine enables you to assert the alignment of a variable at a certain point in the program flow. Specifically, at the call point to **ALIGNX**, you can assert that the remainder from dividing the address of the second argument by the value of the first argument is zero. In case the second argument is a Fortran 90 pointer, the assertion refers to the address of the target. In case the second argument is an integer pointer, the assertion refers to the address of the pointee. Should you give the compiler incorrect alignment, the resulting program may not run correctly if alignment-sensitive instructions are either executed (such as **VMX** operations) or inserted by the optimizer.

Class

Subroutine

Argument type and attributes

K   An INTENT(IN) INTEGER(4). The actual argument corresponding to **K** must be a positive constant expression with a value that is a power of two.

M   A variable of any type. When the actual argument corresponding to **M** is a Fortran 90 pointer, the pointer must be associated.

Examples

```
INTEGER*4 B(200)
  DO N=1, 200
    CALL ALIGNX(4, B(N))   !ASSERTS THAT AT THIS POINT,
    B(N) = N               !B(N) IS 4-BYTE ALIGNED
  END DO
END


SUBROUTINE VEC(A, B, C)
  INTEGER A(200), B(200), C(200)
  CALL ALIGNX(16, A(1))
  CALL ALIGNX(16, B(1))
  CALL ALIGNX(16, C(1))
  DO N = 1, 200
    C(N) = A(N) + B(N)
  END DO
END SUBROUTINE
```

ALL(MASK, DIM)

Purpose

Determines if all values in an entire array, or in each vector along a single dimension, are true.

Class

Transformational function

Argument type and attributes

MASK
An INTENT(IN) LOGICAL array

DIM (optional)
An INTENT(IN) scalar INTEGER. Its value must be in the range $1 \leq$ **DIM** $\leq$ rank(MASK). The corresponding actual argument must not be an optional dummy argument.

Result value

The result is a logical array with the same type parameters as **MASK**. The rank of the result is rank(**MASK**)-1 if the **DIM** is specified; otherwise the result is a scaler of type logical.

The shape of the result is $(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n)$, where n is the rank of MASK.

Each element in the result array is .TRUE. only if all the elements given by **MASK**$(m_1, m_2, ..., m_{(DIM-1)}, :, m_{(DIM+1)}, ..., m_n)$, are true. When the result is a scalar, either because **DIM** is not specified or because MASK is of rank one, it is **.TRUE.** only if all elements of MASK are true, or **MASK** has size zero.

Examples

```
! A is the array | 4 3 6 |, and B is the array | 3 5 2 |
!                | 2 4 1 |                      | 7 8 4 |

! Is every element in A less than the
! corresponding one in B?
      RES = ALL(A .LT. B)          ! result RES is false

! Are all elements in each column of A less than the
! corresponding column of B?
      RES = ALL(A .LT. B, DIM = 1) ! result RES is (f,t,f)

! Same question, but for each row of A and B.
      RES = ALL(A .LT. B, DIM = 2) ! result RES is (f,t)
```

ALLOCATED(X)

Purpose

Indicates whether or not an allocatable object is allocated.

Class

Inquiry function

Argument type and attributes

X can be one of the following:

ARRAY
An allocatable array whose allocation status you want to know.

SCALAR
An allocatable scalar whose allocation status you want to know.

Result type and attributes

Default logical scalar.

Result value

The result corresponds to the allocation status of ARRAY or SCALAR: .TRUE. if it is allocated, .FALSE. if it is not allocated, or undefined if its allocation status is undefined. If you are compiling with the **-qxlf90=autodealloc** compiler option there is no undefined allocation status.

Examples

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
PRINT *, ALLOCATED(A)      ! A is not allocated yet.
ALLOCATE (A(1000))
PRINT *, ALLOCATED(A)      ! A is now allocated.
END
```

Related information

"Allocatable arrays" on page 81, "ALLOCATE" on page 287, "Allocation status" on page 28.

ANINT(A, KIND)
Purpose

Nearest whole number.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) REAL

KIND (optional)
An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- The result type is real.
- If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the kind type parameter is that of A.

Result value

- If A > 0, ANINT(A) = AINT(A + 0.5)
- If A ≤ 0, ANINT(A) = AINT(A - 0.5)

Note: The addition and subtraction of 0.5 are done in round-to-zero mode.

Examples

```
ANINT(3.555) = 4.0
ANINT(-3.555) = -4.0
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ANINT	default real	default real	yes
DNINT	double precision real	double precision real	yes
QNINT **1**	REAL(16)	REAL(16)	yes

Note:

1. IBM extension.

ANY(MASK, DIM)
Purpose

Determines if any of the values in an entire array, or in each vector along a single dimension, are true.

Class

Transformational function

Argument type and attributes

MASK
> An INTENT(IN) LOGICAL array

DIM (optional)
> An INTENT(IN) INTEGER scalar. Its value must be in the range
> 1 ≤ **DIM** ≤ rank(MASK). The corresponding actual argument must not be an optional dummy argument.

Result value

The result is a logical array of the same type parameters as **MASK**. The rank of the result is rank(**MASK**)-1 if the **DIM** is specified; otherwise the result is a scalar of type logical.

The shape of the result is $(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n)$, where n is the rank of MASK.

Each element in the result array is .TRUE. if any of the elements given by MASK($m_1, m_2, ..., m_{(DIM-1)}, :, m_{(DIM+1)}, ..., m_n$) are true. When the result is a scalar, either because DIM is not specified or because MASK is of rank one, it is .TRUE. if any of the elements of MASK are true.

Examples

```
! A is the array | 9 -6 7 | , and B is the array | 2 7 8 |
!                | 3 -1 5 |                       | 5 6 9 |

! Is any element in A greater than or equal to the
! corresponding element in B?
      RES = ANY(A .GE. B)          ! result RES is true

! For each column in A, is there any element in the column
! greater than or equal to the corresponding element in B?
      RES = ANY(A .GE. B, DIM = 1) ! result RES is (t,f,f)

! Same question, but for each row of A and B.
      RES = ANY(A .GE. B, DIM = 2) ! result RES is (t,f)
```

ASIN(X)

Purpose

Arcsine (inverse sine) function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL` F2008 or `COMPLEX`. F2008 . Its value must satisfy the inequality $|X| \leq 1$,

Result type and attributes

Same as X.

Result value

If X is of type real, the result value is as follows:

- It is expressed in radians, and approximates arcsin(X).
- It is in the range $-\pi/2 \leq \text{ASIN(X)} \leq \pi/2$.

F2008

If X is of type complex, the real part of the result value is as follows:

- It is expressed in radians.
- It is in the range $-\pi/2 \leq \text{REAL(ASIN(X))} \leq \pi/2$.

F2008

Examples

ASIN(1.0) approximates $\pi/2$.

F2008 **ASIN**((0.841471, 0.000000)) has the value (1.000000, 0.000000), approximately. F2008

Specific Name	Argument Type	Result Type	Pass As Arg?
ASIN	default real	default real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
DASIN	double precision real	double precision real	yes
QASIN **1**	REAL(16)	REAL(16)	yes
QARSIN **1**	REAL(16)	REAL(16)	yes

Note:

1. IBM extension.

ASIND(X) (IBM extension)
Purpose

Arcsine (inverse sine) function. Result in degrees.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL. Its value must satisfy the inequality $|X| \leq 1$.

Result type and attributes

Same as X.

Result value

- It is expressed in degrees, and approximates arcsin(X).
- It is in the range $-90° \leq ASIND(X) \leq 90°$.

Examples

ASIND (0.5) has the value 30.0°.

Specific Name	Argument Type	Result Type	Pass As Arg?
ASIND	default real	default real	yes
DASIND	double precision real	double precision real	yes
QASIND	REAL(16)	REAL(16)	yes

ASINH(X) (Fortran 2008)
Purpose

Inverse hyperbolic sine function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL or COMPLEX

Result type and attributes

Same as X.

Result value

The result value approximates the inverse hyperbolic cosine of X.

If X is of type complex, the imaginary part of the result value is as follows:
- It is expressed in radians.
- It is in the range $0 \leq AIMAG(ASINH(X)) \leq \pi$.

Examples

ASINH(1.1752012) has the value 1.0, approximately.

ASINH((1.175201, 0.000000)) has the value (1.000000, 0.000000), approximately.

ASSOCIATED(POINTER, TARGET)
Purpose

Returns the association status of its pointer argument, or indicates whether the pointer is associated with the target.

Class

Inquiry function

Argument type and attributes

POINTER
> A pointer of any type whose association status you want to test. Its association status must not be undefined.

TARGET (optional)
> A pointer or target of any type that might or might not be associated with **POINTER**. The association status must not be undefined.

Result type and attributes

Default logical scalar.

Result value

If only the **POINTER** argument is specified, the result is .TRUE. if it is associated with any target and .FALSE. otherwise. If **TARGET** is also specified, the procedure tests whether **POINTER** is associated with **TARGET**, or with the same object that **TARGET** is associated with (if **TARGET** is also pointer).

If a **POINTER** and a **TARGET** of a different shape are associated, this intrinsic will return .FALSE..

If **TARGET** is present, then the result is .FALSE. if one of the following occurs:
- **POINTER** is associated with a zero-sized array.
- **TARGET** is associated with a zero-sized array.

- **TARGET** is a zero-sized array.

Objects with different types or shapes cannot be associated with each other.

Arrays with the same type and shape but different bounds can be associated with each other.

Examples

```
REAL, POINTER, DIMENSION(:,:) :: A
REAL, TARGET, DIMENSION(5,10) :: B, C

NULLIFY (A)
PRINT *, ASSOCIATED (A)    ! False, not associated yet

A => B
PRINT *, ASSOCIATED (A)    ! True, because A is
                           ! associated with B

PRINT *, ASSOCIATED (A,C) ! False, A is not
                           ! associated with C
END
```

ATAN(X)

Purpose

Arctangent (inverse tangent) function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL` `F2008` or `COMPLEX`.`F2008`

Result type and attributes

Same as X.

Result value

If X is of type real, the result value is as follows:
- It is expressed in radians and approximates arctan(X).
- It is in the range $-\pi/2 \leq ATAN(X) \leq \pi/2$.

`F2008`

If X is of type complex, the real part of the result value is as follows:
- It is expressed in radians.
- It is in the range $-\pi/2 \leq REAL(ATAN(X)) \leq \pi/2$.

`F2008`

Examples

ATAN(1.0) approximates $\pi/4$.

F2008 **ATAN**((1.557408, 0.000000)) has the value (1.000000, 0.000000), approximately. **F2008**

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN	default real	default real	yes
DATAN	double precision real	double precision real	yes
QATAN **1**	REAL(16)	REAL(16)	yes

Note:

1. IBM extension.

Related functions

- **F2008** ATAN(Y, X) **F2008**

ATAN(Y, X) (Fortran 2008)
Purpose

Arctangent (inverse tangent) function.

Class

Elemental function

Argument type and attributes

Y   An INTENT(IN) REAL

X   An INTENT(IN) argument of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result type and attributes

Same as X.

Result value

The result is the same as the result of "ATAN2(Y, X)."

Related functions

- "ATAN(X)" on page 586

ATAN2(Y, X)
Purpose

Arctangent (inverse tangent) function. The result is the principal value of the nonzero complex number (X, Y) formed by the real arguments Y and X.

Class

Elemental function

Argument type and attributes

Y   An INTENT(IN) REAL

X An INTENT(IN) argument of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result type and attributes

Same as X.

Result value
- It is expressed in radians and has a value equal to the principal value of the argument of the complex number (X, Y).
- It is in the range $-\pi <$ ATAN2(Y, X) $\leq \pi$.
- If X $\neq$ 0, the result approximates arctan(Y/X).
- If Y $>$ 0, the result is positive.
- If Y $<$ 0, the result is negative.
- If X $=$ 0, the absolute value of the result is $\pi/2$.

The **-qxlf2003=signdzerointr** option controls whether you get Fortran 2003 behavior. See **qxlf2003** in the *XL Fortran Compiler Reference*

- If Y $=$ 0 and X $<$ 0, the result is π.

- If Y $=$ 0 and X $>$ 0, the result is zero.

F2003

- If Y $=$ 0 and X $<$ 0, the result is π if Y is positive real zero and $-\pi$ if Y is negative real zero.

- If Y $=$ 0 and X $>$ 0, the result is Y.

F2003

Examples

ATAN2 (1.5574077, 1.0) has the value 1.0.

Given that:

```
Y = |  1    1 |         X = | -1   1 |
    | -1   -1 |             | -1   1 |
```

the value of ATAN2(Y,X) is approximately:

```
ATAN2 (Y, X) = |  3π/4    π/4 |
               | -3π/4   -π/4 |
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN2	default real	default real	yes
DATAN2	double precision real	double precision real	yes
QATAN2 [1]	REAL(16)	REAL(16)	yes

ATAN2D(Y, X) (IBM extension)
Purpose

Arctangent (inverse tangent) function. The result is the principal value of the nonzero complex number (X, Y) formed by the real arguments Y and X.

Class

Elemental function

Argument type and attributes

Y   An INTENT(IN) REAL

X   An INTENT(IN) argument of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result type and attributes

Same as X.

Result value
- It is expressed in degrees and has a value equal to the principal value of the argument of the complex number (X, Y).
- It is in the range -180° < ATAN2D(Y,X) ≤ 180°.
- If X≠0, the result approximates arctan(Y/X).
- If Y>0, the result is positive.
- If Y<0, the result is negative.
- If Y=0 and X>0, the result is zero.
- If Y=0 and X<0, the result is 180°.
- If X=0, the absolute value of the result is 90°.

Examples

ATAN2D (1.5574077, 1.0) has the value 57.295780181° (approximately).

Given that:
```
Y = |  1.0   1.0 |    X = | -1.0   1.0 |
    | -1.0  -1.0 |        | -1.0   1.0 |
```

then the value of ATAN2D(Y,X) is:
```
ATAN2D(Y,X) =  |  135.0000000°    45.00000000° |
               | -135.0000000°   -45.00000000° |
```

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAN2D	default real	default real	yes
DATAN2D	double precision real	double precision real	yes
QATAN2D	REAL(16)	REAL(16)	yes

ATAND(X) (IBM extension)
Purpose

Arctangent (inverse tangent) function. Result in degrees.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

- It is expressed in degrees and approximates arctan(X).
- It is in the range -90° ≤ ATAND(X) ≤ 90°.

Examples

ATAND (1.0) has the value 45.0°.

Specific Name	Argument Type	Result Type	Pass As Arg?
ATAND	default real	default real	yes
DATAND	double precision real	double precision real	yes
QATAND	REAL(16)	REAL(16)	yes

ATANH(X) (Fortran 2008)
Purpose

Inverse hyperbolic tangent function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL or COMPLEX

Result type and attributes

Same as X.

Result value

The result value approximates the inverse hyperbolic tangent of X.

If X is of type complex, the imaginary part of the result value is as follows:
- It is expressed in radians.
- It is in the range $-\pi/2$ ≤ AIMAG(ATANH(X)) ≤ $\pi/2$.

Examples

ATANH(0.76159416) has the value 1.0, approximately.

ATANH((0.761594, 0.000000)) has the value (1.000000, 0.000000), approximately.

BTEST(I, POS)
Purpose

Tests a bit of an integer value.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

POS An `INTENT(IN)` `INTEGER`. Its value must be nonnegative and be less than BIT_SIZE(I).

Result type and attributes

The result is of type default logical.

Result value

The result has the value .TRUE. if bit POS of I has the value 1 and the value .FALSE. if bit POS of I has the value 0.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

BTEST (8, 3) has the value **.TRUE.**.

```
If A has the value
    | 1 2 |
    | 3 4 |

the value of BTEST (A, 2) is
    | false false |
    | false true  |

and the value of BTEST (2, A) is
    | true  false |
    | false false |
```

See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
BTEST **1**	any integer	default logical	yes

Note:

1. IBM extension.

BIT_SIZE(I)

Purpose

Returns the number of bits in an integer type. Because only the type of the argument is examined, the argument need not be defined.

Class

Inquiry function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

Result type and attributes

Scalar integer with the same kind type parameter as I.

Result value

The result is the number of bits in the integer data type of the argument:

IBM

```
   type                 bits
-----------           ------
 integer(1)                8
 integer(2)               16
 integer(4)               32
 integer(8)               64
```

◀ **IBM**

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

BIT_SIZE (1_4) has the value 32, because the integer type with kind 4 (that is, a four-byte integer) contains 32 bits.

CEILING(A, KIND)
Purpose

Returns the least integer greater than or equal to its argument.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN)` `REAL`

KIND (optional)
     An `INTENT(IN)` scalar `INTEGER`. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- It is of type integer.
- If KIND is present, the kind type parameter is that specified by KIND; otherwise, the **KIND** type parameter is that of the default integer type.

Result value

The result has a value equal to the least integer greater than or equal to A.

The result is undefined if the result cannot be represented as an integer of the specified KIND.

Examples

```
CEILING(-3.7) has the value -3.
CEILING(3.7) has the value 4.

CEILING(1000.1, KIND=2) has the value 1001, with a kind
type parameter of two.
```

CHAR(I, KIND)
Purpose

Returns the character in the given position of the collating sequence associated with the specified kind type parameter. It is the inverse of the function **ICHAR**.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER. Its value must be in the range `IBM` $0 \leq I \leq 127$.
      `IBM`

KIND (optional)
      An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- Character of length one.
- If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the kind type parameter is that of the default character type.

Result value

- The result is the character in position I of the collating sequence associated with the specified kind type parameter.
- **ICHAR (CHAR (I, KIND (C)))** must have the value I for $0 \leq I \leq 127$ and CHAR (ICHAR (C), **KIND** (C)) must have the value C for any representable character.

Examples

`IBM`   **CHAR** (88) has the value 'X'.

Specific Name	Argument Type	Result Type	Pass As Arg?
CHAR	any integer	default character	yes **1**

Notes:

1. IBM extension: the ability to specify a non-default integer argument.

XL Fortran supports only the ASCII collating sequence. `IBM`

CMPLX(X, Y, KIND)
Purpose

Convert to complex type.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER, REAL, COMPLEX, BINARY, OCTAL, or hexadecimal CONSTANT.

Y (optional)
An INTENT(IN) INTEGER, REAL, COMPLEX, BINARY, OCTAL, or hexadecimal CONSTANT. It must not be present if X is a COMPLEX.

KIND (optional)
An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- It is of type complex.
- If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the kind type parameter is that of the default real type.

Result value

- If Y is absent and X is not complex, it is as if Y were present with the value zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).
- CMPLX(X, Y, KIND) has the complex value whose real part is REAL(X, KIND) and whose imaginary part is REAL(Y, KIND).

Examples

CMPLX (-3) has the value (-3.0, 0.0).

Specific Name	Argument Type	Result Type	Pass As Arg?
CMPLX **1**	default real	default complex	no

Note:
1. IBM extension.

Related information

"DCMPLX(X, Y) (IBM extension)" on page 605, "QCMPLX(X, Y) (IBM extension)" on page 682.

COMMAND_ARGUMENT_COUNT() (Fortran 2003)
Purpose

Returns the number of command line arguments for the command that invoked the program.

Class

Inquiry function

Result type and attributes

Default integer scalar

Result value

The result value is the number of command arguments, not counting the command name. If there are no command arguments, the result value is 0.

Examples

```
integer cmd_count
cmd_count = COMMAND_ARGUMENT_COUNT()
print*, cmd_count
end
```

The following is sample output generated by the above program:

```
$ a.out
0
$ a.out aa
1
$ a.out aa bb
2
```

CONJG(Z)
Purpose

Conjugate of a complex number.

Class

Elemental function

Argument type and attributes

Z   An INTENT(IN) COMPLEX

Result type and attributes

Same as Z.

Result value

Given Z has the value (x, y), the result has the value (x, -y).

Examples

CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

Specific Name	Argument Type	Result Type	Pass As Arg?
CONJG	default complex	default complex	yes
DCONJG **1**	double complex	double complex	yes
QCONJG **1**	COMPLEX(16)	COMPLEX(16)	yes

Note:

1. IBM extension.

COS(X)
Purpose

Cosine function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL or COMPLEX

Result type and attributes

Same as X.

Result value
- It has a value that approximates cos(X).
- If X is of type real, X is regarded as a value in radians.
- If X is of type complex, the real and imaginary parts of X are regarded as values in radians.

Examples

COS (1.0) has the value 0.54030231 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
COS	default real	default real	yes
DCOS	double precision real	double precision real	yes
QCOS [1]	REAL(16)	REAL(16)	yes
CCOS [2a]	default complex	default complex	yes
CDCOS [1] [2b]	double complex	double complex	yes
ZCOS [1] [2b]	double complex	double complex	yes
CQCOS [1] [2b]	COMPLEX(16)	COMPLEX(16)	yes

Note:

1. IBM extension.
2. Given that X is a complex number in the form a + bi, where i = $(-1)^{\frac{1}{2}}$:
   a. abs(b) must be less than or equal to 88.7228; a is any real value.
   b. abs(b) must be less than or equal to 709.7827; a is any real value.

COSD(X) (IBM extension)
Purpose

Cosine function. Argument in degrees.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN) REAL`

Result type and attributes

Same as X.

Result value

It approximates cos(X), where X has a value in degrees.

Examples

COSD (45.0°) has the value 0.7071067691.

Specific Name	Argument Type	Result Type	Pass As Arg?
COSD	default real	default real	yes
DCOSD	double precision real	double precision real	yes
QCOSD	REAL(16)	REAL(16)	yes

COSH(X)

Purpose

Hyperbolic cosine function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN) REAL` `F2008` or `COMPLEX`. `F2008`

Result type and attributes

Same as X.

Result value

The result value approximates cosh(X).

`F2008` If X is of type complex, its imaginary part is considered a value in radians. `F2008`

Examples

COSH(1.0) has the value 1.5430806, approximately.

`F2008` **COSH**((1.000000, 0.000000)) has the value (1.543081, 0.000000), approximately. `F2008`

Specific Name	Argument Type	Result Type	Pass As Arg?
COSH	default real	default real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
DCOSH	double precision real	double precision real	yes
QCOSH **1**	REAL(16)	REAL(16)	yes

Note:

1. IBM extension.

COUNT(MASK, DIM, KIND)

Purpose

Counts the number of true array elements in an entire logical array, or in each vector along a single dimension. Typically, the logical array is one that is used as a mask in another intrinsic.

Class

Transformational function

Argument type and attributes

MASK

An INTENT(IN) LOGICAL array

DIM (optional)

An INTENT(IN) INTEGER scalar. Its value must be in the range $1 \leq$ **DIM** $\leq$ rank(**MASK**). The corresponding actual argument must not be an optional dummy argument.

F2003 **KIND (optional)**

An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression.

Result value

If **DIM** is present, the result is an integer array of rank rank(**MASK**)-1. If **DIM** is missing, or if **MASK** has a rank of one, the result is a scalar of type integer.

F2003 If **KIND** is present, the kind of the result is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. F2003

Each element of the resulting array ($R(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n)$) equals the number of elements that are true in **MASK** along the corresponding dimension ($s_1, s_2, ..., s_{(DIM-1)}, :, s_{(DIM+1)}, ..., s_n$).

If **MASK** is a zero-sized array, the result equals zero.

Examples

```
! A is the array | T F F |, and B is the array | F F T |
!                | F T T |                      | T T T |

! How many corresponding elements in A and B
! are equivalent?
    RES = COUNT(A .EQV. B)        ! result RES is 3

! How many corresponding elements are equivalent
! in each column?
```

```
      RES = COUNT(A .EQV. B, DIM=1) ! result RES is (0,2,1)

! Same question, but for each row.
      RES = COUNT(A .EQV. B, DIM=2) ! result RES is (1,2)
```

CPU_TIME(TIME) (Fortran 95)
Purpose

Returns the CPU time, in seconds, taken by the current process and, possibly, all the child processes in all of the threads. A call to **CPU_TIME** will give the processor time taken by the process from the start of the program. The time measured only accounts for the amount of time that the program is actually running, and not the time that a program is suspended or waiting.

Class

Subroutine

Argument type and attributes

TIME An `INTENT(OUT)` REAL scalar. It is assigned an approximation to the processor time. The time is measured in seconds. The time returned by **CPU_TIME** is dependent upon the setting of the **XLFRTEOPTS** environment variable or run-time option **cpu_time_type**. The valid settings for **cpu_time_type** are:

usertime
> The user time for the current process.

systime
> The system time for the current process.

alltime
> The sum of the user and system time for the current process

total_usertime
> The total user time for the current process. The total user time is the sum of the user time for the current process and the total user times for its child processes, if any.

total_systime
> The total system time for the current process. The total system time is the sum of the system time for the current process and the total system times for its child processes, if any.

total_alltime
> The total user and system time for the current process. The total user and system time is the sum of the user and system time for the current process and the total user and system times for their child processes, if any.
>
> This is the default measure of time for **CPU_TIME** if you have not set the **cpu_time_type** run-time option.

You can set the **cpu_time_type** run-time option using the **setrteopts** procedure. Each change to the **cpu_time_type** setting will affect all subsequent calls to **CPU_TIME**.

Examples

Example 1:
```
! The default value for cpu_time_type is used
REAL T1, T2
...          ! First chunk of code to be timed
CALL CPU_TIME(T1)
...          ! Second chunk of code to be timed
CALL CPU_TIME(T2)
print *, 'Time taken for first chunk of code: ', T1, 'seconds.'
print *, 'Time taken for both chunks of code: ', T2, 'seconds.'
print *, 'Time for second chunk of code was ', T2-T1, 'seconds.'
```

If you want to set the **cpu_time_type** run-time option to **usertime**, you would type the following command from a ksh or bsh command line:

```
export XLFRTEOPTS=cpu_time_type=usertime
```

Example 2:
```
! Use setrteopts to set the cpu_time_type run-time option as many times
! as you need to
CALL setrteopts ('cpu_time_type=alltime')
CALL stallingloop
CALL CPU_TIME(T1)
print *, 'The sum of the user and system time is', T1, 'seconds'.
CALL setrteopts ('cpu_time_type=usertime')
CALL stallingloop
CALL CPU_TIME(T2)
print *, 'The total user time from the start of the program is', T2, 'seconds'.
```

Related information

- See the description of the **XLFRTEOPTS** environment variable in the *XL Fortran Compiler Reference* for more information.

CSHIFT(ARRAY, SHIFT, DIM)
Purpose

Shifts the elements of all vectors along a given dimension of an array. The shift is circular; that is, elements shifted off one end are inserted again at the other end.

Class

Transformational function

Argument type and attributes

ARRAY
> An array of any type.

SHIFT
> - An INTENT(IN) INTEGER scalar, if **ARRAY** has a rank of one
> - An INTENT(IN) INTEGER scalar or an INTEGER expression of rank rank(**ARRAY**)-1, if **ARRAY** does not have a rank of one.

DIM (optional)
> An INTENT(IN) INTEGER scalar. Its value must be in the range $1 \leq \text{DIM} \leq \text{rank}(\text{ARRAY})$. If absent, it defaults to 1.

Result value

The result is an array with the same shape, data type, and type parameters as **ARRAY**.

If **SHIFT** is a scalar, the same shift is applied to each vector. Otherwise, each vector **ARRAY** $(s_1, s_2, ..., s_{(DIM-1)}, :, s_{(DIM+1)}, ..., s_n)$ is shifted according to the corresponding value in **SHIFT** $(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n)$

The absolute value of **SHIFT** determines the amount of shift. The sign of **SHIFT** determines the direction of the shift:

Positive SHIFT
> moves each element of the vector toward the beginning of the vector.

Negative SHIFT
> moves each element of the vector toward the end of the vector.

Zero SHIFT
> does no shifting. The value of the vector remains unchanged.

Examples

```
! A is the array │ A D G │
!                │ B E H │
!                │ C F I │

! Shift the first column down one, the second column
! up one, and leave the third column unchanged.
      RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 1)
! The result is │ C E G │
!               │ A F H │
!               │ B D I │

! Do the same shifts as before, but on the rows
! instead of the columns.
      RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 2)
! The result is │ G A D │
!               │ E H B │
!               │ C F I │
```

CVMGx(TSOURCE, FSOURCE, MASK) (IBM extension)
Purpose

The conditional vector merge functions (**CVMGM**, **CVMGN**, **CVMGP**, **CVMGT**, and **CVMGZ**) enable you to port existing code that contains these functions.

Calling them is very similar to calling
```
MERGE ( TSOURCE, FSOURCE, arith_expr .op. 0 )
or
MERGE ( TSOURCE, FSOURCE, logical_expr .op. .TRUE. )
```

Because the **MERGE** intrinsic is part of the Fortran 90 language, we recommend that you use it instead of these functions for any new programs.

Class

Elemental function

Argument type and attributes

TSOURCE

A scalar or array expression of type **LOGICAL**, **INTEGER**, or **REAL** and any kind except 1.

FSOURCE

A scalar or array expression with the same type and kind type parameters as **TSOURCE**.

MASK

A scalar or array expression of type **INTEGER** or **REAL** (for **CVMGM**, **CVMGN**, **CVMGP**, and **CVMGZ**) or **LOGICAL** (for **CVMGT**), and any kind except 1. If it is an array, it must conform in shape to **TSOURCE** and **FSOURCE**.

If only one of **TSOURCE** and **FSOURCE** is typeless, the typeless argument acquires the type of the other argument. If both **TSOURCE** and **FSOURCE** are typeless, both arguments acquire the type of **MASK**. If **MASK** is also typeless, both **TSOURCE** and **FSOURCE** are treated as default integers. If **MASK** is typeless, it is treated as a default logical for the **CVMGT** function and as a default integer for the other **CVMGx** functions.

Result type and attributes

Same as **TSOURCE** and **FSOURCE**.

Result value

The function result is the value of either the first argument or second argument, depending on the result of the test performed on the third argument. If the arguments are arrays, the test is performed for each element of the **MASK** array, and the result may contain some elements from **TSOURCE** and some elements from **FSOURCE**.

Table 60. Result values for CVMGx intrinsic procedures

Explanation	Function Return Value	Generic Name
Test for positive or zero	**TSOURCE** if **MASK**≥0 **FSOURCE** if **MASK**<0	CVMGP
Test for negative	**TSOURCE** if **MASK**<0 **FSOURCE** if **MASK**≥0	CVMGM
Test for zero	**TSOURCE** if **MASK**=0 **FSOURCE** if **MASK**≠0	CVMGZ
Test for nonzero	**TSOURCE** if **MASK**≠0 **FSOURCE** if **MASK**=0	CVMGN
Test for true	**TSOURCE** if **MASK**= **.true.** **FSOURCE** if MASK= **.false.**	CVMGT

DATE_AND_TIME(DATE, TIME, ZONE, VALUES)
Purpose

Returns data from the real-time clock and the date in a form compatible with the representations defined in ISO 8601:1988.

Class

Subroutine

Argument type and attributes

DATE (optional)

An `INTENT(OUT)` `CHARACTER` scalar that has a length of at least eight. Its leftmost eight characters are set to a value of the form **CCYYMMDD**, where **CC** is the century, **YY** is the year within the century, **MM** is the month within the year, and **DD** is the day within the month. If no date is available, these characters are set to blank.

TIME (optional)

An `INTENT(OUT)` `CHARACTER` scalar that has a length of at least ten. Its leftmost ten characters are set to a value of the form hhmmss.sss, where hh is the hour of the day, mm is the minutes of the hour, and ss.sss is the seconds and milliseconds of the minute. If no clock is available, they are set to blank.

ZONE (optional)

An `INTENT(OUT)` `CHARACTER` scalar that has a length of at least five. Its leftmost five characters are set to a value of the form ±hhmm, where hh and mm are the time difference with respect to Coordinated Universal Time (UTC) in hours and the parts of an hour expressed in minutes, respectively. If no clock is available, they are set to blank.

> **IBM** The value of **ZONE** may be incorrect if you have not set the time zone on your hardware correctly. You can manually set the **TZ** environment variable to ensure the time zone is correct. **IBM**

VALUES (optional)

An `INTENT(OUT)` `INTEGER` of rank one. Its size must be at least eight. The values returned in VALUES are as follows:

VALUES(1)

is the year (for example, 1998), or -HUGE (0) if no date is available.

VALUES(2)

is the month of the year, or -HUGE (0) if no date is available.

VALUES(3)

is the day of the month, or -HUGE (0) if no date is available.

VALUES(4)

is the time difference with respect to Coordinated Universal Time (UTC) in minutes, or -HUGE (0) if this information is not available.

VALUES(5)

is the hour of the day, in the range 0 to 23, or -HUGE (0) if there is no clock.

VALUES(6)

is the minutes of the hour, in the range 0 to 59, or -HUGE (0) if there is no clock.

VALUES(7)

is the seconds of the minute, in the range 0 to 60, or -HUGE (0) if there is no clock.

VALUES (8)

is the milliseconds of the second, in the range 0 to 999, or -HUGE (0) if there is no clock.

Examples

The following program:

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
                    BIG_BEN (3), DATE_TIME)
```

if executed in Geneva, Switzerland on 1985 April 12 at 15:27:35.5, would have assigned the value 19850412 to **BIG_BEN**(1), the value 152735.500 to **BIG_BEN**(2), the value +0100 to **BIG_BEN**(3), and the following values to **DATE_TIME**: 1985, 4, 12, 60, 15, 27, 35, 500.

Note that UTC is defined by CCIR Recommendation 460-2 (also known as Greenwich Mean Time).

DBLE(A)
Purpose

Convert to double precision real type.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER, REAL, COMPLEX, or boz-literal CONSTANT

Result type and attributes

Double precision real.

Result value

- If A is of type double precision real, DBLE(A) = A.
- If A is of type integer or real, the result has as much precision of the significant part of A as a double precision real datum can contain.
- If A is of type complex, the result has as much precision of the significant part of the real part of A as a double precision real datum can contain.

Examples

DBLE (-3) has the value -3.0D0.

IBM

Specific Name	Argument Type	Result Type	Pass As Arg?
DFLOAT	any integer	double precision real	no
DBLE	default real	double precision real	no

Specific Name	Argument Type	Result Type	Pass As Arg?
DBLEQ	REAL(16)	REAL(8)	no

IBM

DCMPLX(X, Y) (IBM extension)

Purpose

Convert to double complex type.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER, REAL, or COMPLEX

Y (optional)
An INTENT(IN) INTEGER or REAL. It must not be present if X is of type complex.

Result type and attributes

It is of type double complex.

Result value

• If Y is absent and X is not complex, it is as if Y were present with the value of zero.
• If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).
• DCMPLX(X, Y) has the complex value whose real part is REAL(X, KIND=8) and whose imaginary part is REAL(Y, KIND=8).

Examples

DCMPLX (-3) has the value (-3.0D0, 0.0D0).

Specific Name	Argument Type	Result Type	Pass As Arg?
DCMPLX	double precision real	double complex	no

Related information

"CMPLX(X, Y, KIND)" on page 593, "QCMPLX(X, Y) (IBM extension)" on page 682.

DIGITS(X)

Purpose

Returns the number of significant digits for numbers whose type and kind type parameter are the same as the argument.

Class

Inquiry function

Argument type and attributes

X   An INTENT(IN) INTEGER or REAL. The actual argument corresponding to **X** can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

- **IBM** If X is of type integer, the number of the significant digits of X is:

```
    type                    bits
  -----------              ------
   integer(1)                 7
   integer(2)                15
   integer(4)                31
   integer(8)                63
```

- If X is of type real, the number of significant bits of X is:

```
    type                    bits
  -----------              ------
   real(4)                   24
   real(8)                   53
   real(16)                 106
```

IBM

Examples

IBM **DIGITS** (X) = 63, where X is of type integer(8) (see "Data representation models" on page 570). **IBM**

DIM(X, Y)
Purpose

The difference X-Y if it is positive; otherwise zero.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER or REAL

Y   An INTENT(IN) argument of the same type and kind type parameter as X.

Result type and attributes

Same as X.

Result value

- If X > Y, the value of the result is X - Y.
- If X ≤ Y, the value of the result is zero.

Examples

DIM (-3.0, 2.0) has the value 0.0. **DIM** (-3.0, -4.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
IDIM	any integer **1**	same as argument	yes
DIM	default real	default real	yes
DDIM	double precision real	double precision real	yes
QDIM **2**	REAL(16)	REAL(16)	yes

Note:
1. IBM extension: the ability to specify a nondefault integer argument.
2. IBM extension.

DOT_PRODUCT(VECTOR_A, VECTOR_B)
Purpose

Computes the dot product on two vectors.

Class

Transformational function

Argument type and attributes

VECTOR_A
An `INTENT(IN)` `NUMERIC` or `LOGICAL` vector

VECTOR_B
The same type and size as **VECTOR_A**.

Result value

The result is a scalar whose data type depends on the data type of the two vectors, according to the rules in Table 17 on page 107 and Table 18 on page 111.

If either vector is a zero-sized array, the result equals zero when it has a numeric data type, and false when it is of type logical.

If **VECTOR_A** is of type integer or real, the result value equals SUM(**VECTOR_A** * **VECTOR_B**).

If **VECTOR_A** is of type complex, the result equals SUM(CONJG(**VECTOR_A**) * **VECTOR_B**).

If **VECTOR_A** is of type logical, the result equals ANY(**VECTOR_A** .AND. **VECTOR_B**).

Examples

```
! A is (/ 3, 1, -5 /), and B is (/ 6, 2, 7 /).
      RES = DOT_PRODUCT (A, B)
! calculated as
!   ( (3*6) + (1*2) + (-5*7) )
! = (   18  +   2   +  (-35) )
! =  -15
```

DPROD(X, Y)
Purpose

Double precision real product.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL`

Y   An `INTENT(IN)` `REAL`

Result type and attributes

Double precision real.

Result value

The result has a value equal to the product of X and Y.

Examples

DPROD (-3.0, 2.0) has the value -6.0D0.

Specific Name	Argument Type	Result Type	Pass As Arg?
DPROD	default real	double precision real	yes
QPROD **1**	double precision real	REAL(16)	yes

Note:

1. IBM extension.

DSHIFTL(I, J, SHIFT) (Fortran 2008)
Purpose

Performs a combined shift to the left.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`

J   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`

       If both I and J are of type `INTEGER`, they must be the same kind. I and J must not both be boz-literal `CONSTANT`.

SHIFT

       An `INTENT(IN)` `INTEGER`. Its value must be non-negative, and less than or equal to BIT_SIZE(I) when I is an `INTEGER`. Otherwise, it must be less than or equal to BIT_SIZE(J).

Result type and attributes

Same as I when I is of type integer. Otherwise, same as J.

Result value

- If I or J is a boz-literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The rightmost SHIFT bits of the result value are equal to the leftmost SHIFT bits of J, and other bits of the result value are equal to the rightmost bits of I, which is equal to IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT SIZE (J)-SHIFT)).
- No vacated bits.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Example

DSHIFTL (5, 2**15, 10) has the result 5120.

DSHIFTR(I, J, SHIFT) (Fortran 2008)
Purpose

Performs a combined shift to the right.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`

J   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`

          If both I and J are of type `INTEGER`, they must be the same kind. I and J must not both be boz-literal `CONSTANT`.

SHIFT
          An `INTENT(IN)` `INTEGER`. Its value must be non-negative, and less than or equal to BIT_SIZE(I) when I is an `INTEGER`. Otherwise, it must be less than or equal to BIT_SIZE(J).

Result type and attributes

Same as I when I is of type integer. Otherwise, same as J.

Result value

- If I or J is a boz-literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The leftmost SHIFT bits of the result value are equal to the rightmost SHIFT bits of I, and other bits of the result value are equal to the leftmost bits of J, which is equal to IOR (SHIFTL (I, BIT SIZE (I)-SHIFT), SHIFTR (J, SHIFT)).
- No vacated bits.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Example

DSHIFTR (-50, -128, 30) has the result -197.

EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

Purpose

Shifts the elements of all vectors along a given dimension of an array. The shift is end-off; that is, elements shifted off one end are lost, and copies of boundary elements are shifted in at the other end.

Class

Transformational function

Argument type and attributes

ARRAY
An array of any type.

SHIFT
- An `INTENT(IN)` `INTEGER` scalar, if **ARRAY** has a rank of 1
- Otherwise, an `INTENT(IN)` `INTEGER` scalar or an `INTENT(IN)` expression of rank rank(**ARRAY**)-1

BOUNDARY (optional)
The same type and type parameters as **ARRAY**. If **ARRAY** has a rank of 1, **BOUNDARY** must be scalar. Otherwise, **BOUNDARY** is a scalar or an expression of rank=rank(**ARRAY**)-1, and of shape (d1, d2..., dDIM-1, dDIM+1..., dn).

DIM (optional)
An `INTEGER` scalar. Its value must be in the range $1 \leq$ **DIM** $\leq$ rank(**ARRAY**).

Result value

The result is an array with the same shape, data type, and type parameters as **ARRAY**.

The absolute value of **SHIFT** determines the amount of shift. The sign of **SHIFT** determines the direction of the shift:

Positive SHIFT
moves each element of the vector toward the beginning of the vector. If an element is taken off the beginning of a vector, its value is replaced by the corresponding value from **BOUNDARY** at the end of the vector.

Negative SHIFT
moves each element of the vector toward the end of the vector. If an element is taken off the end of a vector, its value is replaced by the corresponding value from **boundary** at the beginning of the vector.

Zero SHIFT
does no shifting. The value of the vector remains unchanged.

Result value

If **BOUNDARY** is a scalar value, this value is used in all shifts.

If **BOUNDARY** is an array of values, the values of the array elements of **BOUNDARY** with subscripts (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) are used for that dimension.

If **BOUNDARY** is not specified, the following default values are used, depending on the data type of **ARRAY**:

character
> ' ' (one blank)

logical
> false

integer
> 0

real   0.0

complex
> (0.0, 0.0)

Examples

```
! A is | 1.1 4.4 7.7 |,
SHIFT is S=(/0, -1, 1/),
!      | 2.2 5.5 8.8 |
!      | 3.3 6.6 9.9 |
! and BOUNDARY is the array B=(/-0.1, -0.2, -0.3/).

! Leave the first column alone, shift the second
! column down one, and shift the third column up one.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 1)
! The result is | 1.1 -0.2  8.8 |
!               | 2.2  4.4  9.9 |
!               | 3.3  5.5 -0.3 |

! Do the same shifts as before, but on the
! rows instead of the columns.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 2)
! The result is |  1.1  4.4  7.7 |
!               | -0.2  2.2  5.5 |
!               |  6.6  9.9 -0.3 |
```

EPSILON(X)
Purpose

Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An `INTENT(IN)` REAL. The corresponding actual argument can be scalar or array valued.

Result type and attributes

Scalar of the same type and kind type parameter as X.

Result value

The result is

$$2.0 \, ei \, 0^{1 - \text{DIGITS}(X)}$$

where *ei* is the exponent indicator (E, D, or Q) depending on the type of X:

```
type                    EPSILON(X)
----                    ----------
real(4)                 02E0 ** (-23)
real(8)                 02D0 ** (-52)
real(16)                02Q0 ** (-105)
```

Examples

EPSILON (X) = 1.1920929E-07 for X of type real(4). See "Real data model" on page 571.

ERF(X) (Fortran 2008)
Purpose

Error function.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^X e^{-t^2} dt$$

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

- The result value approximates erf(X).
- The result is in the range -1 ≤ ERF(X) ≤ 1.

Examples

ERF (1.0) has the value 0.8427007794 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ERF	default real	default real	yes
DERF [1]	double precision real	double precision real	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
QERF **1**	REAL(16)	REAL(16)	yes
Note: • **1** IBM extension			

ERFC(X) (Fortran 2008)

Purpose

Complementary error function.

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL

Result type and attributes

Same as X.

Result value

- The result has a value equal to 1-ERF(X).
- The result is in the range $0 \le \mathrm{ERFC}(X) \le 2$.

Examples

ERFC (1.0) has the value 0.1572992057 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ERFC	default real	default real	yes
DERFC **1**	double precision real	double precision real	yes
QERFC **1**	REAL(16)	REAL(16)	yes
Note: • **1** IBM extension			

ERFC_SCALED(X) (Fortran 2008)

Purpose

Scaled complementary error function.

$$\mathrm{erfc\_scaled}(x) = \exp(x^2)\mathrm{erfc}(x) = e^{x^2}\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$$

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL`

Result type and attributes

Same as X.

Result value

The result value approximates erfc_scaled(X).

Examples

ERFC_SCALED (1.0) has the value 0.4275836 (approximately).

Related information
- "EXP(X)" on page 616
- "ERFC(X) (Fortran 2008)" on page 613

EXECUTE_COMMAND_LINE(COMMAND, WAIT, EXITSTAT, CMDSTAT, CMDMSG) (Fortran 2008)
Purpose

Passes a command to the operating system for execution.

Class

Subroutine

Argument type and attributes

COMMAND
> An `INTENT(IN)` `CHARACTER` scalar. It specifies the command line to be executed.

WAIT (optional)
> An `INTENT(IN)` `LOGICAL` scalar. It determines whether **COMMAND** is executed synchronously or asynchronously.
> - If **WAIT** is set to .TRUE., **COMMAND** is executed synchronously.
> - If **WAIT** is set to .FALSE., **COMMAND** is executed asynchronously if that is supported, and synchronously otherwise.
> - If **WAIT** is not present, XL Fortran treats it as present with the value .TRUE.. Then **COMMAND** is executed synchronously.
>
> **Note:** In the current release, XL Fortran supports only synchronous execution.

EXITSTAT (optional)

> An INTENT(OUT) INTEGER scalar. If XL Fortran executes **COMMAND** synchronously, **EXITSTAT** is assigned the value of the system return code returned by the executed command when this argument is present. Otherwise, **EXITSTAT** remains unchanged.

CMDSTAT (optional)

> An INTENT(OUT) INTEGER scalar. If **CMDSTAT** is present, it is assigned a value indicating the status of the command execution as follows:
>
> - If command-line execution is not supported, **CMDSTAT** is assigned the value of -1.
> - If no error occurs during the execution of **COMMAND**, and the value of **WAIT** is .FALSE., but asynchronous execution is not supported, **CMDSTAT** is assigned the value of -2.
> - If an error occurs, **CMDSTAT** is assigned the value of 1.
> - In all other cases, **CMDSTAT** is assigned the value of 0.
>
> **Note:** If a condition occurs that would assign a nonzero value to **CMDSTAT**, but the **CMDSTAT** argument is not present, error termination is initiated.

CMDMSG (optional)

> An INTENT(OUT) CHARACTER scalar. If an error occurs during the execution of **COMMAND**, **CMDMSG** is assigned the value of an explanatory message when this argument is present. Otherwise, **CMDMSG** remains unchanged.

Examples

```
CALL EXECUTE_COMMAND_LINE('rm -rf script')
OPEN(1, FILE = 'script')
CALL EXECUTE_COMMAND_LINE('date | awk \'{printf \" %s \", $1}\' >> script.out')

! Execute the 'cmd' command synchronously
CHARACTER(LEN = 10) :: cmd = "du -s -m"
CALL EXECUTE_COMMAND_LINE(cmd, .TRUE.)

! Assign 'exit' the value of the system return code
INTEGER :: exit = 0
CALL EXECUTE_COMMAND_LINE('/bin/mv mod1.mod mod2.mod', .TRUE., EXIT)

! Assign 'exit' the value of the system return code
! Assign 'cmdstat' the value of the status of the command execution
INTEGER :: exit, cmdstat = 0
CALL EXECUTE_COMMAND_LINE('echo running on = \'hostname\'', .TRUE., exit, cmdstat)

! If an error occurs, 'message' is assigned the value of
! an explanatory message.

IMPLICIT NONE

CHARACTER(200) :: message = ""
INTEGER :: exit_s = 0, cmd_s = 0

CALL EXECUTE_COMMAND_LINE("mv f.1 f.2", EXITSTAT = exit_s,
                          CMDSTAT = cmd_s, CMDMSG = message)

IF (cmd_s .NE. 0) THEN
  IF (cmd_s .EQ. -1) THEN
    PRINT *, "command execution not supported on this system"
    STOP 1
  END IF

  PRINT *, message
  STOP 2
END IF
```

```
    IF (exit_s .NE. 0) THEN
      STOP 3
    END IF

    END
```

EXP(X)

Purpose

Exponential.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL` or `COMPLEX`

Result type and attributes

Same as X.

Result value

- The result approximates e^x.
- If X is of type complex, its real and imaginary parts are regarded as values in radians.

Examples

EXP (1.0) has the value 2.7182818 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
EXP **1**	default real	default real	yes
DEXP **2**	double precision real	double precision real	yes
QEXP **2** **3**	REAL(16)	REAL(16)	yes
CEXP **4a**	default complex	default complex	yes
CDEXP **4b** **3**	double complex	double complex	yes
ZEXP **4b** **3**	double complex	double complex	yes
CQEXP **4b** **3**	COMPLEX(16)	COMPLEX(16)	yes

Note:

1. X must be less than or equal to 88.7228.
2. X must be less than or equal to 709.7827.
3. IBM extension.
4. When X is a complex number in the form a + bi, where i = (-1) $^{\frac{1}{2}}$:

    a. a must be less than or equal to 88.7228; b is any real value.

    b. a must be less than or equal to 709.7827; b is any real value.

EXPONENT(X)

Purpose

Returns the exponent part of the argument when represented as a model number.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL

Result type and attributes

Default integer.

Result value

- If X ≠ 0, the result is the exponent of X (which is always within the range of a default integer).
- If X = 0, the exponent of X is zero.

Examples

EXPONENT (10.2) = 4. See "Real data model" on page 571

EXTENDS_TYPE_OF(A, MOLD) (Fortran 2003)

Purpose

Inquires whether the dynamic type of A is an extension type of the dynamic type of MOLD.

Class

Inquiry function

Argument type and attributes

A   An `INTENT(IN)` argument of extensible type. If the actual argument corresponding to **A** is a pointer, the pointer must not have an undefined association status.

MOLD
An `INTENT(IN)` argument of extensible type. If the actual argument corresponding to **MOLD** is a pointer, the pointer must not have an undefined association status.

Result type and attributes

Default logical scalar

Result value

- If **MOLD** is unlimited polymorphic and is a disassociated pointer or an allocatable argument that is deallocated, the result is true.

- Otherwise, if **A** is unlimited polymorphic and is either a disassociated pointer or an allocatable argument that is deallocated, the result is false.
- Otherwise, if the dynamic type of **A** is an extension type of the dynamic type of **MOLD**, the result is true.
- Otherwise, the result is false.

Note: The result depends only on the dynamic types of **A** and **MOLD**. Differences in type parameters are ignored.

FINDLOC(ARRAY, VALUE, DIM, MASK, KIND, BACK) or FINDLOC(ARRAY, VALUE, MASK, KIND, BACK) (Fortran 2008)

Purpose

Locates the first or the last element of an array along a dimension that equals the target value corresponding to the true values of the mask. **FINDLOC** returns the subscript of the element using positive integers.

Class

Transformational function

Argument type and attributes

ARRAY
An array of type integer, real, complex, logical, character, or byte.

VALUE
A scalar of the same type as **ARRAY**.

DIM **(optional)**
An integer scalar. Its value must be in the range $1 \leq DIM \leq n$, where n is the rank of **ARRAY**. The corresponding actual argument cannot be an optional dummy argument.

MASK **(optional)**
An array of type logical and of the same shape as **ARRAY**. If it is absent, the default mask evaluation is `.TRUE.`, which means that the entire array is evaluated.

KIND **(optional)**
An integer scalar. The actual argument corresponding to **KIND** must be a constant expression.

BACK **(optional)**
A logical scalar. It controls the direction in which **ARRAY** is searched. If it is `.FALSE.` or absent, the array is searched from the beginning. If it is `.TRUE.`, the array is searched from the end.

Result type and attributes

- If **KIND** is present, the **KIND** type parameter is that specified by **KIND**; otherwise, the **KIND** type parameter is that of default integer type.
- If **DIM** is absent, the result is an integer array of rank one with the size equal to the rank of **ARRAY**.
- If **DIM** is absent, and if there is no matching value because the **DIM** is zero-sized, or the mask array has all `.FALSE.` values, or there is no element in the **ARRAY** equal to the **VALUE** argument, the return value is a one-dimensional array whose elements are zero.

- If **DIM** is present, the result is an integer array of rank rank(**ARRAY**) -1, and the shape is (s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n), where n is the rank of **ARRAY**.

Restriction: FINDLOC cannot be passed as an actual argument.

Result value

The result indicates the subscript of the location of the masked element of **ARRAY** whose value matches **VALUE**. If **ARRAY** is of type character, the comparison is done using the ASCII collating sequence.

If more than one element equals **VALUE**, and **BACK** is absent or has the value .FALSE., the result indicates the location of the first element in array element order. If **BACK** is present with the value .TRUE., the result indicates the location of the last element in array element order.

If **DIM** is specified, the result indicates the location of the masked element of **ARRAY** whose value matches **VALUE** along each vector of the dimension. Example 3 demonstrates the results according to different value of **DIM** in a 2-rank array.

Examples

Example 1:
```
PRINT *, FINDLOC([4,9,-2,9], VALUE=9)
```

It prints | 2 | because the array is searched from the beginning and the second element is the first one that is of target value.
```
PRINT *, FINDLOC([4,9,-2,9], VALUE=9, BACK=.TRUE.)
```

It prints | 4 | because **BACK** has the value .TRUE. and the fourth element is the last one that is of target value.

Example 2:
```
INTEGER :: A(-2:0,5:8)      ! The lower bound of A does not affect the result.
LOGICAL :: M(-5:-3,-1:2)    ! A and M can have different lower bounds.
A = RESHAPE((/3,2,7,8,5,1,-4,1,0,5,3,5/), (/3,4/))
M = RESHAPE((/.FALSE., .TRUE., .FALSE., .TRUE., .FALSE., .TRUE., .TRUE.,
            .TRUE., .TRUE., .TRUE., .TRUE., .TRUE./), (/3,4/))
!A has the value |3  8 -4  5|, and M has the value |.FALSE. .TRUE.  .TRUE. .TRUE.|.
!                |2  5  1  3|                       |.TRUE.  .FALSE. .TRUE. .TRUE.|
!                |7  1  0  5|                       |.FALSE. .TRUE.  .TRUE. .TRUE.|
```

To find the first element of value 5 using mask M:
```
PRINT *, FINDLOC(A, 5, MASK=M)
```

It prints | 1 4 | because the target value, 5, is at A(1, 4). A(2, 2), which also has target value, is not evaluated because of mask M. Although A(3,4) is also of target value, it is not the first in column-major order.

To find the last element of value 5 using mask M:
```
PRINT *, FINDLOC(A, 5, MASK=M, BACK=.TRUE.)
```

It prints | 3 4 | because the target value, 5, is at A(3, 4). Although there are other instances of target value, A(3, 4) is the last in column-major order.

Example 3:

```
INTEGER  :: B(2,3)
B = RESHAPE((/6,4,-2,3,4,5/),(/2,3/))
!B has the value |6 -2  4|.
!                |4  3  5|
```

To find the first element of value 4 in each column of B, specify *DIM* = 1:

```
PRINT *, FINDLOC(B, VALUE=4, DIM=1)
```

It prints | 2 0 1|. These numbers are the corresponding row locations of the first element that is of the target value, 4, in each column. The found elements are at B(2, 1) and B(1, 3). There is no matching value in the second column, so 0 is in the second place.

To find the first element of value 4 in each row of B, specify *DIM* = 2:

```
PRINT *, FINDLOC(B, VALUE=4, DIM=2)
```

It prints | 3 1|. These numbers are the corresponding column locations of the first element that is of target value in each row. The found elements are at B(1, 3) and B(2, 1).

FLOOR(A, KIND)

Purpose

Returns the greatest integer less than or equal to its argument.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) REAL

KIND (optional)
An INTENT(IN) scalar INTEGER. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

It is of type integer.

If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the **KIND** type parameter is that of the default integer type.

Result value

The result has a value equal to the greatest integer less than or equal to **A**.

The result is undefined if the result cannot be represented as an integer of the specified **KIND**.

Examples

```
FLOOR(-3.7) has the value -4.
FLOOR(3.7) has the value 3.
```

FLOOR(1000.1, KIND=2) has the value 1000, with a kind type parameter of two.

FRACTION(X)

Purpose

Returns the fractional part of the model representation of the argument value.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL`

Result type and attributes

Same as X.

Result value

IBM The result is:

```
X * (2.0-EXPONENT(X))
```

IBM

Examples

IBM

```
FRACTION(10.2) =2-4 * 10.2 approximately equal to 0.6375
```

IBM

GAMMA(X) (Fortran 2008)

Purpose

Gamma function.

$$\Gamma(x) = \int_0^\infty u^{x-1} e^{-u} du$$

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL`

Result type and attributes

Same as X.

Result value

The result has a value that approximates $\Gamma(X)$.

Examples

GAMMA(1.0) has the value 1.0.

GAMMA(10.0) has the value 362880.0, approximately.

Specific Name	Argument Type	Result Type	Pass As Arg?
GAMMA **1**	default real	default real	yes
DGAMMA **2** **4**	double precision real	double precision real	yes
QGAMMA **3** **4**	REAL(16)	REAL(16)	yes

Notes:
• X must satisfy the inequality:
    1 -2.0**23 < X ≤ 35.0401, except for nonpositive integral values
    2 -2.0**52 < X ≤ 171.6243, except for nonpositive integral values
    3 -2.0**105 < X ≤ 171.6243, except for nonpositive integral values
• **4** IBM extension			

GET_COMMAND(COMMAND, LENGTH, STATUS) (Fortran 2003)
Purpose

Returns the command that invoked the program.

Class

Subroutine

Argument type and attributes

COMMAND (optional)
> An INTENT(OUT) CHARACTER scalar. It is the command that invoked the program, or a string of blanks if the command is unknown.

LENGTH (optional)
> An INTENT(OUT) INTEGER scalar. It is the significant length of the command that invoked the program, or 0 if the length of the command is unknown. This length includes significant trailing blanks of each argument. It does not include any truncation or padding that occurs when the command is assigned to the **COMMAND** argument.

STATUS (optional)
> An INTENT(OUT) INTEGER scalar. It is a status value.

> **STATUS** has one of the following values:

> • 1 if the command retrieval fails

> • -1 if the **COMMAND** argument is present and has a length less than the significant length of the command

> • 0 otherwise

Examples

```
integer len, status
character(7) :: cmd
call    GET_COMMAND(cmd, len, status)
print*, cmd
print*, len
print*, status
end
```

The following code is sample output the above program generates:

```
$ a.out
a.out      (followed by two spaces)
5
0
$ a.out aa
a.out a
8
-1
```

GET_COMMAND_ARGUMENT(NUMBER, VALUE, LENGTH, STATUS) (Fortran 2003)

Purpose

Returns a command line argument of the command that invoked the program.

Class

Subroutine

Argument type and attributes

NUMBER

An INTENT(IN) INTEGER scalar. It is an integer that identifies the argument number. 0 represents the command name. The numbers from 1 to the argument count represent the command's arguments.

VALUE (optional)

An INTENT(OUT) CHARACTER scalar. It is assigned the value of the argument, or a string of blanks if the value is unknown.

LENGTH (optional)

An INTENT(OUT) INTEGER scalar. It is assigned the significant length of the argument, or 0 if the length of the argument is unknown. This length includes significant trailing blanks. It does not include any truncation or padding that occurs when the argument is assigned to the VALUE argument.

STATUS (optional)

An INTENT(OUT) INTEGER scalar. It is assigned a status value.

It has one of the following values:

- 1 if the argument retrieval fails
- -1 if the VALUE argument is present and has a length less than the significant length of the command argument
- 0 otherwise

Examples

```
integer num, len, status
character*7 value
num = 0
call GET_COMMAND_ARGUMENT(num, value, len, status)
print*, value
print*, len
print*, status
```

See the following sample output generated by the above program:

```
$ a.out aa bb
a.out     (followed by two spaces)
5
0
```

GET_ENVIRONMENT_VARIABLE(NAME, VALUE, LENGTH, STATUS, TRIM_NAME) (Fortran 2003)

Purpose

Returns the value of the specified environment variable.

Class

Subroutine

Argument type and attributes

NAME
>   An INTENT(IN) CHARACTER scalar. It is a character string that identifies the name of the operating-system environment variable. The string is case-significant.

VALUE (optional)
>   An INTENT(OUT) CHARACTER scalar. It is assigned the value of the environment variable, or a string of blanks if the environment variable has no value or does not exist.

LENGTH (optional)
>   An INTENT(OUT) INTEGER scalar. It is assigned the significant length of the value, or 0 if the environment variable has no value or does not exist.

STATUS (optional)
>   An INTENT(OUT) INTEGER scalar. It is assigned a status value.
>
>   **STATUS** has one of the following values:
>   - 0, if either the environment variable exists and its value is successfully assigned to **VALUE** or the environment variable exists but has no value
>   - 1, if the environment variable does not exist
>   - -1, if the length of the **VALUE** argument is less than the significant length of the value of the environment variable
>   - 3, if other error conditions occur

TRIM_NAME (optional)
>   An **INTENT(IN) LOGICAL** scalar. **TRIM_NAME** is a logical value that specifies whether to trim trailing blanks in **NAME**. By default, trailing blanks in **NAME** are trimmed. If **TRIM_NAME** exists and has the value .FALSE., trailing blanks in **NAME** are considered significant.

Examples

```
integer num, len, status
character*15 value
call GET_ENVIRONMENT_VARIABLE('HOME', value, len, status)
print*, value
print*, len
print*, status
```

The following is sample output generated by the above program:

```
$ a.out
/home/xlfuser      (followed by two spaces)
13
0
```

GETENV(NAME, VALUE) (IBM extension)
Purpose

Returns the value of the specified environment variable.

Note: This is an IBM extension. It is recommended that you use the GET_ENVIRONMENT_VARIABLE intrinsic procedure for portability.

Class

Subroutine

Argument type and attributes

NAME

An `INTENT(IN)` `CHARACTER` scalar. Its value is a character string that identifies the name of the operating-system environment variable. The string is case-significant.

VALUE

An `INTENT(OUT)` `CHARACTER` scalar. It holds the value of the environment variable when the subroutine returns.

Result value

The result is returned in the VALUE argument, not as a function result variable.

If the environment variable specified in the NAME argument does not exist, the VALUE argument contains blanks.

Examples
```
      CHARACTER (LEN=16)   ENVDATA
      CALL GETENV('HOME', VALUE=ENVDATA)
! Print the value.
      PRINT *, ENVDATA
! Show how it is blank-padded on the right.
      WRITE(*, '(Z32)') ENVDATA
      END
```

The following is sample output generated by the above program:
```
/home/mark
2F686F6D652F6D61726B202020202020
```

HFIX(A) (IBM extension)
Purpose

Convert from **REAL(4)** to **INTEGER(2)**.

This procedure is a specific function, not a generic function.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN) REAL(4)`

Result type and attributes

An **INTEGER(2)** scalar or array.

Result value
- If |A| < 1, HFIX (A) has the value 0.
- If |A| ≥ 1, HFIX (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.
- The result is undefined if the result cannot be represented in an **INTEGER(2)**.

Examples

HFIX (-3.7) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
HFIX	REAL(4)	INTEGER(2)	no

HYPOT(X, Y) (Fortran 2008)
Purpose

Calculates the Euclidean distance between two values.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN) REAL`

Y   An `INTENT(IN)` argument of the same type and kind type parameter as X.

Result type and attributes

Same as X.

Result value

The result value is equal to sqrt($x^2 + y^2$), approximately.

Example

HYPOT(3.0, 4.0) has the value 5.0.

HUGE(X)

Purpose

Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An INTENT(IN) INTEGER or REAL

Result type and attributes

Scalar of the same type and kind type parameter as X.

Result value

- If X is of any integer type, the result is:

  $2^{\text{DIGITS}(X)} - 1$

- If X is of any real type, the result is:

  $(1.0 - 2.0^{-\text{DIGITS}(X)}) * (2.0^{\text{MAXEXPONENT}(X)})$

Examples

IBM

HUGE (X) = (1D0 - 2D0**-53) * (2D0**1024) for X of type real(8).

HUGE (X) = (2**63) - 1 for X of type integer(8).

See "Data representation models" on page 570.

IBM

IACHAR(C, KIND)

Purpose

Returns the position of a character in the ASCII collating sequence.

Class

Elemental function

Argument type and attributes

C   An INTENT(IN) character of length one.

F2003 **KIND (optional)**
An INTENT(IN) INTEGER scalar. The actual argument corresponding to **KIND** must be a constant expression. **F2003**

Result type and attributes

- It is of type integer.

- F2003 If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. F2003

Result value

- If C is in the collating sequence defined by the codes specified in ISO 646:1983 (International Reference Version), the result is the position of C in that sequence and satisfies the inequality (0 ≤ IACHAR (C) ≤ 127). An undefined value is returned if C is not in the ASCII collating sequence.
- The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, LLE (C, D) is true, so IACHAR (C) .LE. IACHAR (D) is true too.

Examples

IACHAR ('X') has the value 88.

IAND(I, J)

Purpose

Performs a bitwise AND on two integers.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

J   An `INTENT(IN)` `INTEGER` with the same kind type parameter as I.

Result type and attributes

Same as I.

Result value

The result has the value obtained by combining I and J bit-by-bit according to the following table:

```
ith bit  ith bit      ith bit
 of I     of J        of IAND(I,J)
------------------------------
  1        1            1
  1        0            0
  0        1            0
  0        0            0
```

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IAND (1, 3) has the value 1. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IAND [1]	any integer	same as argument	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
AND **1**	any integer	same as argument	yes
Note: **1** IBM extension			

IBCLR(I, POS)

Purpose

Clears one bit to zero.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

POS   An INTENT(IN) INTEGER. Its value must be nonnegative and less than BIT_SIZE (I).

Result type and attributes

Same as I.

Result value

The result has the value of the sequence of bits of I, except that bit POS of I is set to zero.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBCLR (14, 1) has the result 12.

If V has the value (/1, 2, 3, 4/), the value of IBCLR (POS = V, I = 31) is (/29, 27, 23, 15/).

See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBCLR **1**	any integer	same as argument	yes
Note: **1** IBM extension			

IBITS(I, POS, LEN)

Purpose

Extracts a sequence of bits.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

POS   An INTENT(IN) INTEGER. Its value must be nonnegative and POS + LEN
           must be less than or equal to BIT_SIZE (I).

LEN   An INTENT(IN) INTEGER. Its value must be nonnegative.

Result type and attributes

Same as I.

Result value

The result has the value of the sequence of LEN bits in I beginning at bit POS,
right-adjusted and with all other bits zero.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBITS (14, 1, 3) has the value 7. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBITS **1**	any integer	same as argument	yes
Note: **1** IBM extension			

IBM2GCCLDBL(A) (IBM extension)
Purpose

Converts IBM-style long double data types to GCC long doubles.

Class

Elemental function.

Argument type and attributes

A   An INTENT(IN) REAL(16)

Result type and attributes

Same as A.

Result value

The result has the REAL(16) value in A, converted to a REAL(16) value compatible
with GCC's glibc library.

Examples

Specific Name	Argument Type	Result Type	Pass As Arg?
IBM2GCCLDBL **1**	REAL(16)	same as argument	yes

Note:

1. IBM extension.

IBM2GCCLDBL_CMPLX(A)
Purpose

Converts IBM-style long double data types to GCC long doubles.

Class

Elemental function.

Argument type and attributes

A   An `INTENT(IN)` `COMPLEX(16)` .

Result type and attributes

Same as A.

Result value

This result has the COMPLEX(16) value in A, converted to a COMPLEX(16) value compatible with GCC's glibc library.

Examples

Specific Name	Argument Type	Result Type	Pass As Arg?
IBM2GCCLDBL_CMPLX **1**	COMPLEX(16)	same as argument	yes

Note:

1. IBM extension.

IBSET(I, POS)
Purpose

Sets one bit to one.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

POS   An `INTENT(IN)` `INTEGER`. Its value must be nonnegative and less than BIT_SIZE (I).

Result type and attributes

Same as I.

Result value

The result has the value of the sequence of bits of I, except that bit POS of I is set to one.

The bits are numbered from 0 to BIT_SIZE(I)-1, from right to left.

Examples

IBSET (12, 1) has the value 14.

If V has the value (/1, 2, 3, 4/), the value of IBSET (POS = V, I = 0) is (/2, 4, 8, 16/).

See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IBSET **1**	any integer	same as I	yes
Note: **1** IBM extension			

ICHAR(C, KIND)
Purpose

Returns the position of a character in the collating sequence associated with the kind type parameter of the character.

Class

Elemental function

Argument type and attributes

C   An `INTENT(IN)` `CHARACTER` of length one. Its value must be that of a representable character.

F2003 **KIND (optional)**
An `INTENT(IN)` `INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. **F2003**

Result type and attributes

• It is of type integer.
• **F2003** If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. **F2003**

Result value

• The result is the position of C in the collating sequence associated with the kind type parameter of C and is in the range 0 ≤ ICHAR (C) ≤ 127.

- For any representable characters C and D, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C) .EQ. ICHAR (D) is true.

Examples

IBM **ICHAR** ('X') has the value 88 in the ASCII collating sequence.

Specific Name	Argument Type	Result Type	Pass As Arg?
ICHAR	default character	default integer	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument. **2** XL Fortran supports only the ASCII collating sequence.			

IBM

IEOR(I, J)

Purpose

Performs an exclusive OR.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN) INTEGER`

J   An `INTENT(IN) INTEGER` with the same kind type parameter as I.

Result type and attributes

Same as I.

Result value

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

```
ith bit  ith bit    ith bit
 of I     of J    of IEOR(I,J)
-----------------------------
   1        1          0
   1        0          1
   0        1          1
   0        0          0
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

IEOR (1, 3) has the value 2. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IEOR **1**	any integer	same as argument	yes
XOR **1**	any integer	same as argument	yes

Specific Name	Argument Type	Result Type	Pass As Arg?
Note: **1** IBM extension			

ILEN(I) (IBM extension)
Purpose

Returns one less than the length, in bits, of the twos complement representation of an integer.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

Result type and attributes

Same as I.

Result value
- If I is negative, ILEN(I)=CEILING(LOG2(-I))
- If I is nonnegative, ILEN(I)=CEILING(LOG2(I+1))

Examples
```
I=ILEN(4)  ! 3
J=ILEN(-4) ! 2
```

IMAG(Z) (IBM extension)
Purpose

Identical to **AIMAG**.

Related information

"AIMAG(Z), IMAG(Z)" on page 577.

INDEX(STRING, SUBSTRING, BACK, KIND)
Purpose

Returns the starting position of a substring within a string.

Class

Elemental function

Argument type and attributes

STRING
     An `INTENT(IN)` `CHARACTER`

SUBSTRING
> An INTENT(IN) CHARACTER with the same kind type parameter as STRING.

BACK (optional)
> An INTENT(IN) LOGICAL

F2003 **KIND (optional)**
> An INTENT(IN) INTEGER scalar. The actual argument corresponding to KIND must be a constant expression. F2003

Result type and attributes

- It is of type integer.
- F2003 If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. F2003

Result value

- Case (i): If BACK is absent or present with the value .FALSE., the result is the minimum positive value of I such that STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING). One is returned if LEN (SUBSTRING) = 0.
- Case (ii): If BACK is present with the value .TRUE., the result is the maximum value of I less than or equal to LEN (STRING) - LEN (SUBSTRING) + 1, such that STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples

INDEX ('FORTRAN', 'R') has the value 3.

INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

Specific Name	Argument Type	Result Type	Pass As Arg?
INDEX	default character	default integer	yes **1**
Note: **1** When this specific name is passed as an argument, the procedure can only be referenced without the **BACK** and **KIND** optional argument.			

INT(A, KIND)

Purpose

Convert to integer type.

Class

Elemental function

Argument type and attributes

A An INTENT(IN) INTEGER, REAL, COMPLEX, or boz-literal CONSTANT

KIND (optional)
> An INTENT(IN) INTEGER scalar. The actual argument corresponding to KIND must be a constant expression. F2003

Result type and attributes

- Integer.
- If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the kind type parameter is that of the default integer type.

Result value

- Case (i): If A is of type integer, INT (A) = A.
- Case (ii): If A is of type real, there are two cases: if |A| < 1, INT (A) has the value 0; if |A| ≥ 1, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.
- Case (iii): If A is of type complex, INT (A) is the value obtained by applying the case (ii) rule to the real part of A.
- Case (iv): If A is a boz-literal constant, it is treated as an integer with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor. If **-qxlf2003=nobozlitargs** is specified the boz-literal is treated as a real.
- The result is undefined if it cannot be represented in the specified integer type.

Examples

INT (-3.7) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
INT	default real	default integer	no
IDINT	double precision real	default integer	no
IFIX	default real	default integer	no
IQINT **1**	REAL(16)	default integer	no
Note: **1** IBM extension			

Related information

For information on alternative behavior for **INT** when porting programs to XL Fortran, see the **-qport** compiler option in the *XL Fortran Compiler Reference*.

INT2(A) (IBM extension)
Purpose

Converts a real or integer value into a two byte integer.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER or REAL

INT2 cannot be passed as an actual argument of another function call.

Result type and attributes

INTEGER(2) scalar

Result value

If *A* is of type integer, INT2(A) = A.

If *A* is of type real, there are two possibilities:
- If |A| < 1, INT2(A) has the value 0
- If |A| >= 1, INT2(A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *A*, and whose sign is the same as the sign of *A*.

In both cases, truncation may occur.

Examples

The following is an example of the **INT2** function.

```
REAL*4 :: R4
REAL*8 :: R8
INTEGER*4 :: I4
INTEGER*8 :: I8

R4 = 8.8; R8 = 18.9
I4 = 4; I8 = 8
PRINT *, INT2(R4), INT2(R8), INT2(I4), INT2(I8)
PRINT *, INT2(2.3), INT2(6)
PRINT *, INT2(65535.78), INT2(65536.89)
END
```

The following is sample output generated by the program above:

```
8 18 4 8
2 6
-1 0     ! The results indicate that truncation has occurred, since
         ! only the last two bytes were saved.
```

IOR(I, J)
Purpose

Performs an inclusive OR.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

J   An `INTENT(IN)` `INTEGER` with the same kind type parameter as I.

Result type and attributes

Same as I.

Result value

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

```
ith bit  ith bit     ith bit
 of I     of J      of IOR(I,J)
---------------------------
   1        1           1
   1        0           1
   0        1           1
   0        0           0
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

IOR (1, 3) has the value 3. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
IOR **1**	any integer	same as argument	yes
OR **1**	any integer	same as argument	yes
Note: 　　**1** IBM extension			

IS_CONTIGUOUS(ARRAY) (Fortran 2008)
Purpose

Tests contiguity of an array.

Class

Inquiry function

Argument type and attributes

ARRAY
　　　　An **INTENT(IN)** argument of any type. The actual argument corresponding to **ARRAY** must be an array `TS 29113` or an assumed-rank object `TS 29113` of any type. If the actual argument is a pointer, it must be associated.

Result type and attributes

Default logical scalar

Result value

Returns .TRUE. if **ARRAY** is contiguous`TS 29113` or has rank zero`TS 29113`. Otherwise, returns .FALSE.

Examples

```
INTEGER, POINTER :: ap(:)
INTEGER, TARGET :: targ(10)
ap => targ(1:10:2)
PRINT *, IS_CONTIGUOUS(ap)        ! IS_CONTIGUOUS(ap) returns the .FALSE. value.
```

IS_IOSTAT_END(I) (Fortran 2003)

Purpose

Checks for an end-of-file condition.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

Result type and attributes

Default logical scalar.

Result value

Returns .TRUE. if the argument matches the value of the **IOSTAT=** specifier when an end-of-file condition has occurred. Otherwise, **IS_IOSTAT_END** returns .FALSE..

Examples

The following is an example of **IS_IOSTAT_END**:

```
program a
  integer :: ios = 0, x

  open( 1, file='dat.dat', action='read' )

  do while( .not. is_iostat_end(ios) )

      read( 1,*,iostat=ios ) x
      write(6,*) "ios = ", ios
      write(6,*) "x = ", x

   enddo
end program a
```

IS_IOSTAT_EOR(I) (Fortran 2003)

Purpose

Checks for an end-of-record condition.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

Result type and attributes

Default logical scalar

Result value

Returns .TRUE. if the argument matches the value of the **IOSTAT=** specifier when an end-of-record condition has occurred. Otherwise, **IS_IOSTAT_EOR** returns .FALSE..

ISHFT(I, SHIFT)
Purpose

Performs a logical shift.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

SHIFT
> An `INTENT(IN)` `INTEGER`. The absolute value of SHIFT must be less than or equal to BIT_SIZE (I).

Result type and attributes

Same as I.

Result value
- The result has the value obtained by shifting the bits of I by SHIFT positions.
- If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and, if SHIFT is zero, no shift is performed.
- Bits shifted out from the left or from the right, as appropriate, are lost.
- Vacated bits are filled with zeros.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

ISHFT (3, 1) has the result 6. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
ISHFT **1**	any integer	same as argument	yes
Note: **1** IBM extension			

ISHFTC(I, SHIFT, SIZE)
Purpose

Performs a circular shift of the rightmost bits; that is, bits shifted off one end are inserted again at the other end.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` INTEGER

SHIFT

An `INTENT(IN)` INTEGER. The absolute value of SHIFT must be less than or equal to SIZE.

SIZE (optional)

An `INTENT(IN)` INTEGER. The value of SIZE must be positive and must not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Result type and attributes

Same as I.

Result value

The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and, if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

ISHFTC (3, 2, 3) has the value 5. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
ISHFTC **1**	any integer	same as argument	yes **2**
Note: **1** IBM extension **2** When this specific name is passed as an argument, the procedure can only be referenced with all three arguments.			

KIND(X)

Purpose

Returns the value of the kind type parameter of X.

Class

Inquiry function

Argument type and attributes

X   An `INTENT(IN)` argument of any intrinsic type

Result type and attributes

Default integer scalar.

Result value

The result has a value equal to the kind type parameter value of X.

Kind type parameters supported by XL Fortran are defined in Chapter 3, "Intrinsic data types," on page 37.

Examples

KIND (0.0) has the kind type parameter value of the default real type.

LBOUND(ARRAY, DIM, KIND)
Purpose

Returns the lower bound of each dimension in an array, or the lower bound of a specified dimension.

Class

Inquiry function

Argument type and attributes

ARRAY
> The array whose lower bounds you want to determine. The bounds of the array must be defined; that is, the corresponding actual argument cannot be a disassociated pointer or an allocatable array that is not allocated.

DIM (optional)
> An `INTEGER` scalar. Its value must be in the range $1 \le$ **DIM** $\le$ **RANK(ARRAY)**. The corresponding actual argument must not be an optional dummy argument. `TS 29113` The **DIM** argument cannot be present if **ARRAY** is an assumed-rank object that is associated with a scalar. `TS 29113`

`F2003` **KIND (optional)**
> An `INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. `F2003`

Result type and attributes

- The result is of type integer.
- `F2003` If **KIND** is present, the kind type parameter is that specified by the value of **KIND**; otherwise, the kind type parameter is that of the default integer type. `F2003`
- If **DIM** is present, the result is a scalar. If **DIM** is not present, the result is a one-dimensional array with one element for each dimension in **ARRAY**.
- `TS 29113` If **ARRAY** is an assumed-rank object that is associated with a scalar, the result is a zero-sized array. `TS 29113`

Result value

Each element in the result corresponds to a dimension of **array**.

- If **ARRAY** is a whole array or array structure component, **LBOUND(ARRAY, DIM)** is equal to the lower bound for subscript **DIM** of **ARRAY**.

The only exception is for a dimension that is zero-sized and **ARRAY** is not an assumed-size array of rank **DIM**, in such a case, the corresponding element in the result is one regardless of the value declared for the lower bound.

- If **ARRAY** is an array section or expression that is not a whole array or array structure component, each element has the value one.

Examples

```
        REAL A(1:10, -4:5, 4:-5)

        RES=LBOUND( A )
! The result is (/ 1, -4, 1 /).

        RES=LBOUND( A(:,:,:) )
        RES=LBOUND( A(4:10,-4:1,:) )
! The result in both cases is (/ 1, 1, 1 /)
! because the arguments are array sections.
```

LEADZ(I) (Fortran 2008)
Purpose

Returns the number of leading zero bits in the binary representation of an integer.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

Result type and attributes

Same as I.

Result value

The result is the count of zero bits to the left of the leftmost one bit for I. If I has the value zero, the result is **BIT_SIZE(I)**.

Examples

```
I = LEADZ(0_4)  ! I=32
J = LEADZ(4_4)  ! J=29
K = LEADZ(-1)   ! K=0
M = LEADZ(0_8)  ! M=64
N = LEADZ(1_8)  ! N=63
```

Related information
- "BIT_SIZE(I)" on page 591
- "TRAILZ(I) (Fortran 2008)" on page 717

LEN(STRING, KIND)
Purpose

Returns the length of a character entity. The argument to this function need not be defined.

Class

Inquiry function

Argument type and attributes

STRING
> An INTENT(IN) character. The corresponding actual argument can be scalar or array valued. If the actual argument is a deallocated allocatable or a pointer that is not associated, its length type parameter must not be deferred.

F2003 **KIND (optional)**
> An INTENT(IN) INTEGER scalar. The actual argument corresponding to **KIND** must be a constant expression. **F2003**

Result type and attributes

- It is of type scalar integer.
- **F2003** If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. **F2003**

Result value

The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

Examples

If C is declared by the statement

```
CHARACTER (11) C(100)
```

LEN (C) has the value 11.

Specific Name	Argument Type	Result Type	Pass As Arg?
LEN	default character	default integer	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LEN_TRIM(STRING, KIND)
Purpose

Returns the length of the character argument without counting trailing blank characters.

Class

Elemental function

Argument type and attributes

STRING
> An INTENT(IN) CHARACTER

F2003 **KIND (optional)**
An INTENT(IN) INTEGER scalar. The actual argument corresponding to
KIND must be a constant expression. **F2003**

Result type and attributes

- It is of type integer.
- **F2003** If **KIND** is present, the **KIND** type parameter is that specified by the
value of **KIND**; otherwise, the **KIND** type parameter is that of default integer
type. **F2003**

Result value

The result has a value equal to the number of characters remaining after any
trailing blanks in STRING are removed. If the argument contains no nonblank
characters, the result is zero.

Examples

LEN_TRIM (' A B ') has the value 4. **LEN_TRIM** ('   ') has the value 0.

LGAMMA(X) (IBM extension)
Purpose

Log of gamma function.

Class

Elemental function

Argument type and attributes

X An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

The result has a value equal to $\log_e \Gamma(X)$.

Examples

LGAMMA(1.0) has the value 0.0.

LGAMMA(10.0) has the value 12.80182743, approximately.

Related functions

- **F2008** LOG_GAMMA(X) **F2008**

LGE(STRING_A, STRING_B)
Purpose

Test whether a string is lexically greater than or equal to another string, based on
the ASCII collating sequence.

Class

Elemental function

Argument type and attributes

STRING_A
An `INTENT(IN)` CHARACTER

STRING_B
An `INTENT(IN)` CHARACTER

Result type and attributes

Default logical.

Result value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if the strings are equal or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Examples

LGE ('ONE', 'TWO') has the value **.FALSE.**.

Specific Name	Argument Type	Result Type	Pass As Arg?
LGE	default character	default logical	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LGT(STRING_A, STRING_B)
Purpose

Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class

Elemental function

Argument type and attributes

STRING_A
An `INTENT(IN)` CHARACTER

STRING_B
An `INTENT(IN)` CHARACTER

Result type and attributes

Default logical.

Result value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Examples

LGT ('ONE', 'TWO') has the value **.FALSE.**.

Specific Name	Argument Type	Result Type	Pass As Arg?
LGT	default character	default logical	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LLE(STRING_A, STRING_B)
Purpose

Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Class

Elemental function

Argument type and attributes

STRING_A
An INTENT(IN) CHARACTER

STRING_B
An INTENT(IN) CHARACTER

Result type and attributes

Default logical.

Result value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Examples

LLE ('ONE', 'TWO') has the value **.TRUE.**.

Specific Name	Argument Type	Result Type	Pass As Arg?
LLE	default character	default logical	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LLT(STRING_A, STRING_B)
Purpose

Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Class

Elemental function

Argument type and attributes

STRING_A
> An INTENT(IN) CHARACTER

STRING_B
> An INTENT(IN) CHARACTER

Result type and attributes

Default logical.

Result value

- If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.
- If either string contains a character not in the ASCII character set, the result is undefined.
- The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Examples

LLT ('ONE', 'TWO') has the value **.TRUE.**.

Specific Name	Argument Type	Result Type	Pass As Arg?
LLT	default character	default logical	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LOC(X) (IBM extension)
Purpose

Returns the address of X that can then be used to define an integer **POINTER**.

Class

Inquiry function

Argument type and attributes

X   The data object whose address you want to find. The corresponding actual argument must not be an undefined or disassociated pointer or a parameter. `F2008` If the actual argument is an array, it must be contiguous. `F2008` If the actual argument is a zero-sized array, it must be storage associated with a non-zero-sized storage sequence. If the actual argument is an array section, the storage of the array section must be contiguous.

Result type and attributes

The result is of type **INTEGER(8)**.

Result value

The result is the address of the data object, or, if X is a pointer, the address of the associated target. The result is undefined if the argument is not valid.

Examples

```
INTEGER A,B
POINTER (P,I)

P=LOC(A)
P=LOC(B)
END
```

LOG(X)

Purpose

Natural logarithm.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL` or `COMPLEX`
- If X is `REAL`, its value must be greater than zero.
- If X is `COMPLEX`, its value must not be zero.

Result type and attributes

Same as X.

Result value

- It has a value approximating $\log_e X$.
- For complex arguments, LOG ((a,b)) approximates LOG (ABS((a,b))) + ATAN2((b,a)).

The **-qxlf2003=signdzerointr** option controls whether you get Fortran 2003 behavior. See **qxlf2003** in the *XL Fortran Compiler Reference*

-
  If the argument type is complex, the result is the principal value of the imaginary part ω in the range -π < ω ≤ π. If the real part of the argument is less than zero and its imaginary part is zero, the imaginary part of the result approximates π.

-
  F2003 If the argument type is complex, the result is the principal value of the imaginary part ω in the range -π < ω ≤ π. If the real part of the argument is less than zero and its imaginary part is zero, the imaginary part of the result approximates π if the imaginary part of X is positive real zero. If the imaginary part of X is negative real zero, the imaginary part of the result approximates -π F2003

Examples

LOG (10.0) has the value 2.3025851 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
ALOG	default real	default real	yes
DLOG	double precision real	double precision real	yes
QLOG	REAL(16)	REAL(16)	yes **1**
CLOG	default complex	default complex	yes
CDLOG	double complex	double complex	yes **1**
ZLOG	double complex	double complex	yes **1**
CQLOG	COMPLEX(16)	COMPLEX(16)	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LOG_GAMMA(X) (Fortran 2008)
Purpose

Logarithm of the absolute value of the **GAMMA** function.

$$\log_e \Gamma(x) = \log_e \int_0^\infty u^{x-1} e^{-u} du$$

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL. Its value must be greater than 0.

Result type and attributes

Same as X.

Result value

The result value approximates the natural logarithm of the absolute value of the **GAMMA** function of X, namely LOG(ABS(GAMMA(X))).

Examples

LOG_GAMMA(1.0) has the value 0.0, approximately.

Specific Name	Argument Type	Result Type	Pass As Arg?
LGAMMA	default real	default real	no
LGAMMA	double precision real	double precision real	no
ALGAMA **1** **2**	default real	default real	yes
DLGAMA **1** **3**	double precision real	double precision real	yes
QLGAMA **1** **4**	REAL(16)	REAL(16)	yes
Notes:			
• **1** IBM extension			
• X must satisfy the inequality:			
2 $0 < X \leq 4.0850E36$			
3 $2.3561D\text{-}304 \leq X \leq 2^{1014}$			
4 $2.3561Q\text{-}304 \leq X \leq 2^{1014}$			

Related functions

- GAMMA(X)
- **IBM** LGAMMA(X) **IBM**

LOG10(X)

Purpose

Common logarithm.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL. The value of X must be greater than zero.

Result type and attributes

Same as X.

Result value

The result has a value equal to $\log_{10}X$.

Examples

LOG10 (10.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
ALOG10	default real	default real	yes
DLOG10	double precision real	double precision real	yes
QLOG10	REAL(16)	REAL(16)	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

LOGICAL(L, KIND)
Purpose

Converts between objects of type logical with different kind type parameter values.

Class

Elemental function

Argument type and attributes

L   An INTENT(IN) LOGICAL

KIND (optional)
    An INTENT(IN) INTEGER scalar. The actual argument corresponding to
    KIND must be a constant expression. F2003

Result type and attributes

- Logical.
- If **KIND** is present, the kind type parameter is that specified by **KIND**;
  otherwise, the kind type parameter is that of the default logical type.

Result value

The value is that of L.

Examples

LOGICAL (L .OR. .NOT. L) has the value **.TRUE.** and is of type default logical,
regardless of the kind type parameter of the logical variable L.

LSHIFT(I, SHIFT) (IBM extension)
Purpose

Performs a logical shift to the left. It is identical to "SHIFTL(I, SHIFT) (Fortran
2008)" on page 699.

Note: You are encouraged to use the standard intrinsic procedure rather than the
equivalent IBM extension.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

SHIFT
  An `INTENT(IN)` INTEGER. Its value must be non-negative and less than or equal to BIT_SIZE(I).

Result type and attributes

Same as I.

Result value

- The result has the value obtained by shifting the bits of I by SHIFT positions to the left.
- Vacated bits are filled with zeros.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

LSHIFT (3, 1) has the result 6.

LSHIFT (3, 2) has the result 12.

Specific Name	Argument Type	Result Type	Pass As Arg?
LSHIFT	any integer	same as argument	yes

MASKL(I, KIND) (Fortran 2008)
Purpose

Performs a left-justified mask.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` INTEGER. Its value must be non-negative, and less than or equal to the number of bits for the kind of the result.

KIND (optional)
  An `INTENT(IN)` INTEGER scalar. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

The result is an integer. In the presence of KIND argument, its kind is specified by the value of KIND. Otherwise, its kind is of default integer type.

Result value

The leftmost I bits set to 1 and the other bits set to 0.

Example

MASKL (1) has the result -2147483648.

MASKR(I, KIND) (Fortran 2008)
Purpose

Performs a right-justified mask.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER. Its value must be non-negative, and less than or equal to the number of bits for the kind of the result.

KIND (optional)
An INTENT(IN) INTEGER scalar. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

The result is an integer. In the presence of KIND argument, its kind is specified by the value of KIND. Otherwise, its kind is of default integer type.

Result value

The rightmost I bits set to 1 and the other bits set to 0.

Example

MASKR (3) has the result 7.

MATMUL(MATRIX_A, MATRIX_B, MINDIM)
Purpose

Performs a matrix multiplication.

Class

Transformational function

Argument type and attributes

MATRIX_A
An NUMERIC or LOGICAL array with a rank of one or two.

MATRIX_B
An NUMERIC or LOGICAL array with a rank of one or two. It can be a different NUMERIC type than **MATRIX_A**, but you cannot use one NUMERIC matrix and one LOGICAL matrix.

IBM **MINDIM (optional)**
An integer that determines whether to do the matrix multiplication using the Winograd variation of the Strassen algorithm, which may be faster for large matrices. The algorithm recursively splits the operand matrices into four roughly equal parts, until any submatrix extent is less than **MINDIM**.

Note: Strassen's method is not stable for certain row or column scalings of the input matrices. Therefore, for **MATRIX_A** and **MATRIX_B** with divergent exponent values, Strassen's method may give inaccurate results.

The significance of the value of **MINDIM** is:

<=0   does not use the Strassen algorithm at all. This is the default.

1   is reserved for future use.

>1   recursively applies the Strassen algorithm as long as the smallest extent of all dimensions in the argument arrays is greater than or equal to this value. To achieve optimal performance you should experiment with the value of **MINDIM** as the optimal value depends on your machine configuration, available memory, and the size, type, and kind type of the arrays.

By default, **MATMUL** employs the conventional O(N**3) method of matrix multiplication.

If you link the **libpthreads.a** library, the Winograd variation of the O(N**2.81) Strassen method is employed under these conditions:

1. **MATRIX_A** and **MATRIX_B** are both integer, real, or complex and have the same kind.

2. The program can allocate the needed temporary storage, enough to hold approximately (2/3)*(N**2) elements for square matrices of extent N.

3. The **MINDIM** argument is less than or equal to the smallest of all extents of **MATRIX_A** and **MATRIX_B**. **IBM**

At least one of the arguments must be of rank two. The size of the first or only dimension of **MATRIX_B** must be equal to the last or only dimension of **MATRIX_A**.

Result value

The result is an array. If one of the arguments is of rank one, the result has a rank of one. If both arguments are of rank two, the result has a rank of two.

The data type of the result depends on the data type of the arguments, according to the rules in Table 17 on page 107 and Table 18 on page 111.

If **MATRIX_A** and **MATRIX_B** have a numeric data type, the array elements of the result are:

Value of Element (i,j) = SUM((row i of **MATRIX_A**) * (column j of **MATRIX_B**))

If **MATRIX_A** and **MATRIX_B** are of type logical, the array elements of the result are:

Value of Element (i,j) = ANY((row i of **MATRIX_A**) .AND. (column j of **MATRIX_B**))

Examples

```
! A is the array  │ 1 2 3 │, B is the array  │ 7 10 │
!                 │ 4 5 6 │                  │ 8 11 │
!                                            │ 9 12 │
   RES = MATMUL(A, B)
! The result is   │  50    68 │
!                 │ 122   167 │
```

IBM

```
! HUGE_ARRAY and GIGANTIC_ARRAY in this example are
! large arrays of real or complex type, so the operation
! might be faster with the Strassen algorithm.

   RES = MATMUL(HUGE_ARRAY, GIGANTIC_ARRAY, MINDIM=196)
```

IBM

Related information

IBM

The numerical stability of Strassen's method for matrix multiplication is discussed in:

- "Exploiting Fast Matrix Multiplication Within the Level 3 BLAS", Nicholas J. Higham, *ACM Transactions on Mathematical Software*, Vol. 16, No. 4, December 1990.
- "GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm", Douglas, C. C., Heroux, M., Slishman, G., and Smith, R. M., *Journal of Computational Physics*, Vol. 110, No. 1, January 1994, pages 1-10. **IBM**

MAX(A1, A2, A3, ...)
Purpose

Maximum value.

Class

Elemental function

Argument type and attributes

All the arguments must have the same type, either integer, real or character, and they all must have the same kind type parameter.

Result type and attributes

If the arguments are of the type character, the result is of type character, and the length of the result is the length of the longest argument. Otherwise the result type is the same as that of the arguments. (Some specific functions return results of a particular type.)

Result value

The value of the result is that of the largest argument. For character arguments, the comparison is done using the ASCII collating sequence. If the length of the selected

argument is shorter than that of the longest argument, the result is extended to the
length of the longest argument by inserting blank characters on the right.

Examples

MAX (-9.0, 7.0, 2.0) has the value 7.0.

MAX ("Z", "BB") has the value "Z".

Specific Name	Argument Type	Result Type	Pass As Arg?
AMAX0	any integer **1**	default real	no
AMAX1	default real	default real	no
DMAX1	double precision real	double precision real	no
QMAX1	REAL(16)	REAL(16)	no
MAX0	any integer **1**	same as argument	no
MAX1	any real **2**	default integer	no
Note: **1** IBM extension: the ability to specify a nondefault integer argument. **2** IBM extension: the ability to specify a nondefault real argument.			

MAXEXPONENT(X)
Purpose

Returns the maximum exponent in the model representing numbers of the same
type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An `INTENT(IN)` REAL. It can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

IBM See the following result:

```
  type            MAXEXPONENT
----------       -----------
 real(4)              128
 real(8)             1024
 real(16)            1024
```

IBM

Examples

IBM

```
MAXEXPONENT(X) = 128 for X of type real(4).
```

See "Real data model" on page 571.

|IBM|

MAXLOC(ARRAY, DIM, MASK, KIND, BACK) or MAXLOC(ARRAY, MASK, KIND, BACK)

Purpose

Locates the first element `F2008` or the last element `F2008` of an array along a dimension that has the maximum value of all elements corresponding to the true values of the mask. MAXLOC will return the index referable to the position of the element using a positive integer.

Class

Transformational function

Argument type and attributes

ARRAY
An `INTEGER`, `REAL`, or `CHARACTER` array

DIM An `INTEGER` scalar. Its value must be in the range $1 \leq DIM \leq n$, where n is the rank of **ARRAY**.

MASK (optional)
An argument of type `LOGICAL` and conforms to **ARRAY** in shape. If it is absent, the default mask evaluation is `.TRUE.`; that is, the entire array is evaluated.

`F2003` **KIND (optional)**
An `INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. `F2003`

`F2008` **BACK (optional)**
A scalar of type `LOGICAL` and controls the direction in which **ARRAY** is searched. If it is `.FALSE.` or absent, the array is searched from the front. If it is `.TRUE.`, the array is searched from the end. `F2008`

Result type and attributes

- `F2003` If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. `F2003`

- If **DIM** is absent, the result is an integer array of rank one with a size equal to the rank of **ARRAY**. If **DIM** is present, the result is an integer array of rank rank(**ARRAY**)-1, and the shape is $(s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$, where n is the rank of **ARRAY**.

- If there is no maximum value, perhaps because the array is zero-sized or the mask array has all `.FALSE.` values or there is no **DIM** argument, the return value is a zero-sized one-dimensional entity. If **DIM** is present, the result shape depends on the rank of **ARRAY**.

Result value

The result indicates the subscript of the location of the maximum masked element of **ARRAY**. If **ARRAY** is of type character, the comparison is done using the ASCII collating sequence. `F2008` If more than one element is equal to this maximum

value and **BACK** is absent or has the value `.FALSE.`, the function finds the location of the first element in array element order. If **BACK** is present with the value `.TRUE.`, the function finds the location of the last element in array element order. F2008 If **DIM** is specified, the result indicates the location of the maximum masked element along each vector of the dimension.

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the `-qintlog` option is specified with two arguments, the second argument refers to one of the following:

- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless
- **MASK** if it is a scalar of type logical

Examples

```
! A is the array  |  4  9  8 -7 |
!                 |  2  1 -1  5 |
!                 |  9  5 -1  9 |
!                 | -7  5  5 -7 |
```

To find the first largest element of A:
```
RES = MAXLOC(A)
```

The result is | 3 1 | because the maximum value, 9, is located at A(3, 1). Although there are other instances of the maximum value present, A(3, 1) is the first in column-major order.

F2008 To find the last largest element of A:
```
RES = MAXLOC(A, BACK = .TRUE.)
```

The result is | 3 4 | because the maximum value, 9, is located at A(3, 4). Although there are other instances of the maximum value present, A(3, 4) is the last in column-major order. F2008

To find the first largest element that is less than 7 in each column of A:
```
RES = MAXLOC(A, DIM = 1, MASK = A .LT. 7)
```

The result is | 1 3 4 2 | because these are the corresponding row locations of the first largest value that is less than 7 in each column (the values being 4, 5, 5, 5).

F2008 To find the last largest element that is less than 7 in each column of A:
```
RES = MAXLOC(A, DIM = 1, MASK = A .LT. 7, BACK = .TRUE.)
```

The result is | 1 4 4 2 | because these are the corresponding row locations of the last largest value that is less than 7 in each column (the values being 4, 5, 5, 5). F2008

Regardless of the defined upper and lower bounds of the array, MAXLOC will determine the lower bound index as '1'. Both MAXLOC and MINLOC index using positive integers. To find the actual index:
```
        INTEGER B(-100:100)
! Maxloc views the bounds as (1:201)
! If the largest element is located at index '-49'
        I = MAXLOC(B)
! Will return the index '52'
```

```
        ! To return the exact index for the largest element, insert:
              INDEX = LBOUND(B) - 1 + I
        ! Which is:  INDEX = (-100) - 1 + 52 = (-49)
              PRINT*, B(INDEX)
```

MAXVAL(ARRAY, DIM, MASK) or MAXVAL(ARRAY, MASK)

Purpose

Returns the maximum value of the elements in the array along a dimension corresponding to the true elements of MASK.

Class

Transformational function

Argument type and attributes

ARRAY
> An INTEGER, REAL, or CHARACTER array.

DIM (optional)
> An INTEGER scalar. Its value must be in the range
> $1 \le \mathbf{DIM} \le \text{rank(ARRAY)}$.

MASK (optional)
> A LOGICAL array that conforms to **ARRAY** in shape. If it is absent, the entire array is evaluated.

Result value

The result is an array of rank rank(**ARRAY**)-1, with the same data type as **ARRAY**. If **DIM** is missing or if **ARRAY** is of rank one, the result is a scalar. If **ARRAY** is of type character, the length of the result is the same as that of **ARRAY**.

If **DIM** is specified, each element of the result value contains the maximum value of all the elements that satisfy the condition specified by **MASK** along each vector of the dimension **DIM**. The array element subscripts in the result are $(s_1, s_2, ..., s_{(DIM-1)}, s_{(DIM+1)}, ..., s_n)$, where n is the rank of **ARRAY** and DIM is the dimension specified by DIM.

If **DIM** is not specified, the function returns the maximum value of all applicable elements.

If **ARRAY** is of type character, all comparisons are done using the ASCII collating sequence.

If **ARRAY** is zero-sized or the mask array has all .FALSE. values, then:
- if **ARRAY** is of type integer or real, the result value is the negative number of the largest magnitude, of the same type and kind type as **ARRAY**.
- if **ARRAY** is of type character, each character of the result has the value of CHAR(0).

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:
- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless

- **MASK** if it is a scalar of type logical

Examples

```
! A is the array  │ -41  33 25 │
!                 │  12 -61 11 │

! What is the largest value in the entire array?
      RES = MAXVAL(A)
! The result is 33

! What is the largest value in each column?
      RES = MAXVAL(A, DIM=1)
! The result is │ 12 33 25 │

! What is the largest value in each row?
      RES = MAXVAL(A, DIM=2)
! The result is │ 33 12 │

! What is the largest value in each row, considering only
! elements that are less than 30?
      RES = MAXVAL(A, DIM=2, MASK = A .LT. 30)
! The result is │ 25 12 │
```

MERGE(TSOURCE, FSOURCE, MASK)
Purpose

Selects between two values, or corresponding elements in two arrays. A logical mask determines whether to take each result element from the first or second argument.

Class

Elemental function

Argument type and attributes

TSOURCE
> The source array to use when the corresponding element in the mask is true. It is an expression of any data type.

FSOURCE
> The source array to use when the corresponding element in the mask is false. It must have the same data type and type parameters as tsource. It must conform in shape to tsource.

MASK
> A logical expression that conforms to **TSOURCE** and **FSOURCE** in shape.

Result value

The result has the same shape, data type, and type parameters as **TSOURCE** and **FSOURCE**.

For each element in the result, the value of the corresponding element in **MASK** determines whether the value is taken from **TSOURCE** (if true) or FSOURCE (if false).

Examples

```
! TSOURCE is │ A D G │, FSOURCE is │ a d g │,
!            │ B E H │            │ b e h │
!            │ C F I │            │ c f i │
!
! and MASK is the array │ T T T │
!                       │ F F F │
!                       │ F F F │

! Take the top row of TSOURCE, and the remaining elements
! from FSOURCE.
        RES = MERGE(TSOURCE, FSOURCE, MASK)
! The result is │ A D G │
!               │ b e h │
!               │ c f i │

! Evaluate IF (X .GT. Y) THEN
!               RES=6
!           ELSE
!               RES=12
!           END IF
! in a more concise form.
        RES = MERGE(6, 12, X .GT. Y)
```

MERGE_BITS(I, J, MASK) (Fortran 2008)

Purpose

Merges bits under mask.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`

J   An `INTENT(IN)` `INTEGER` or boz-literal `CONSTANT`.

           If both I and J are of type `INTEGER` they must have the same kind type
           parameter. I and J must not both be boz-literal `CONSTANT`.

MASK
           An argument of type integer or a boz-literal constant. If MASK is of type
           integer, its kind must be the same as each other argument of type integer.

Result type and attributes

Same as I if I is of type integer. Otherwise, same as J.

Result value

If any argument is a boz-literal constant, it is first converted to the type and kind
type parameter of the result. The result has the value of IOR (IAND (I,
MASK),IAND (J, NOT (MASK))).

Example

MERGE_BITS (5, 10, 41) has the result 3.

MIN(A1, A2, A3, ...)
Purpose

Minimum value.

Class

Elemental function

Argument type and attributes

All the arguments must have the same type, either integer, real, or character and they all must have the same kind type parameter.

Result type and attributes

If the arguments are of the type character, the result is of type character, and the length of the result is the length of the longest argument. Otherwise, the result is the same as that of the arguments. (Some specific functions return results of a particular type.)

Result value

The value of the result is that of the smallest argument. For character arguments, the comparison is done using the ASCII collating sequence. If the length of the selected argument is shorter than that of the longest argument, the result is extended to the length of the longest argument by inserting blank characters on the right.

Examples

MIN (-9.0, 7.0, 2.0) has the value -9.0

MIN ("A", "YY") has the value "A"

Specific Name	Argument Type	Result Type	Pass As Arg?
AMIN0	any integer **1**	default real	no
AMIN1	default real	default real	no
DMIN1	double precision real	double precision real	no
QMIN1	REAL(16)	REAL(16)	no
MIN0	any integer **1**	same as argument	no
MIN1	any real **1**	default integer	no
Note: **1** A non-default argument is an IBM extension.			

MINEXPONENT(X)
Purpose

Returns the minimum (most negative) exponent in the model representing the numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An INTENT(IN) REAL. The actual argument corresponding to **X** can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

IBM See the following result:

```
   type          MINEXPONENT
----------       -----------
  real(4)            - 125
  real(8)            -1021
  real(16)            -968
```

IBM

Examples

IBM

```
MINEXPONENT(X) = -125 for X of type real(4).
```

See "Real data model" on page 571. **IBM**

MINLOC(ARRAY, DIM, MASK, KIND, BACK) or MINLOC(ARRAY, MASK, KIND, BACK)
Purpose

Locates the first element, **F2008** or the last element if **BACK** is .TRUE., **F2008** of an array along a dimension that has the minimum value of all elements corresponding to the true values of the mask. MINLOC will return the index referable to the position of the element using a positive integer.

Class

Transformational function

Argument type and attributes

ARRAY
       An INTEGER, REAL, or CHARACTER array.

DIM An INTEGER scalar. Its value must be in the range 1≤DIM≤*n*, where *n* is the rank of **ARRAY**.

MASK (optional)
       An argument of type LOGICAL and conforms to **ARRAY** in shape. If it is absent, the default mask evaluation is .TRUE.; that is, the entire array is evaluated.

F2003 KIND (optional)
An `INTENT(IN) INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. **F2003**

F2008 BACK (optional)
A scalar of type `LOGICAL` and controls the direction in which **ARRAY** is searched. If it is `.FALSE.` or absent, the array is searched from the front. If it is `.TRUE.`, the array is searched from the end. **F2008**

Result type and attributes

- **F2003** If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. **F2003**
- If **DIM** is absent, the result is an integer array of rank one with a size equal to the rank of **ARRAY**. If **DIM** is present, the result is an integer array of rank rank(**ARRAY**)-1, and the shape is $(s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$, where n is the rank of **ARRAY**.
- If there is no minimum value, perhaps because the array is zero-sized or the mask array has all `.FALSE.` values or there is no **DIM** argument, the return value is a zero-sized one-dimensional entity. If **DIM** is present, the result shape depends on the rank of **ARRAY**.

Result value

The result indicates the subscript of the location of the minimum masked element of **ARRAY**. If **ARRAY** is of type character, the comparison is done using the ASCII collating sequence. **F2008** If more than one element is equal to this minimum value and **BACK** is absent or has the value `.FALSE.`, the function finds the location of the first element in array element order. If **BACK** is present with the value `.TRUE.`, the function finds the location of the last element in array element order. **F2008** If **DIM** is specified, the result indicates the location of the minimum masked element along each vector of the dimension.

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless
- **MASK** if it is a scalar or type logical

Examples

```
! A is the array  | 4  9  8 -7 |
!                 | 2  1 -1  5 |
!                 | 9  5 -1  9 |
!                 |-7  5  5 -7 |
```

To find the first smallest element of A:
```
RES = MINLOC(A)
```

The result is | 4 1 | because the minimum value, -7, is located at A(4, 1). Although there are other instances of the minimum value present, A(4, 1) is the first in column-major order.

F2008 To find the last smallest element of A:
```
RES = MINLOC(A, BACK = .TRUE.)
```

The result is | 4 4 | because the minimum value, -7, is located at A(4, 4). Although there are other instances of the minimum value present, A(4, 4) is the last in column-major order. `F2008`

To find the first smallest element that is not equal to -7 in each row of A:

```
RES = MINLOC(A, DIM = 2, MASK = A .NE. -7)
```

The result is | 1 3 3 2 | because these are the corresponding column locations of the first smallest value that is not equal to -7 in each row (the values being 4, -1, -1, 5).

`F2008` To find the last smallest element that is not equal to -7 in each row of A:

```
RES = MINLOC(A, DIM = 2, MASK = A .NE. -7, BACK = .TRUE.)
```

The result is | 1 3 3 3 | because these are the corresponding column locations of the last smallest value that is not equal to -7 in each row (the values being 4, -1, -1, 5). `F2008`

Regardless of the defined upper and lower bounds of the array, MINLOC will determine the lower bound index as '1'. Both MAXLOC and MINLOC index using positive integers. To find an actual index:

```
      INTEGER B(-100:100)
! Minloc views the bounds as (1:201)
! If the smallest element is located at index '-49'
      I = MINLOC(B)
! Will return the index '52'
! To return the exact index for the smallest element, insert:
      INDEX = LBOUND(B) - 1 + I
! Which is:  INDEX = (-100) - 1 + 52 = (-49)
      PRINT*, B(INDEX)
```

MINVAL(ARRAY, DIM, MASK) or MINVAL(ARRAY, MASK)

Purpose

Returns the minimum value of the elements in the array along a dimension corresponding to the true elements of MASK.

Class

Transformational function

Argument type and attributes

ARRAY
> An INTEGER, REAL, or CHARACTER array.

DIM (optional)
> An INTEGER scalar. Its value must be in the range
> $1 \le \textbf{DIM} \le \text{rank(ARRAY)}$.

MASK (optional)
> An array or scalar of type LOGICAL that conforms to **ARRAY** in shape. If it is absent, the entire array is evaluated.

Result value

The result is an array of rank rank(**ARRAY**)-1, with the same data type as **ARRAY**. If **DIM** is missing or if **ARRAY** is of rank one, the result is a scalar. If **ARRAY** is of type character, the length of the result is the same as that of **ARRAY**.

If **DIM** is specified, each element of the result value contains the minimum value of all the elements that satisfy the condition specified by **MASK** along each vector of the dimension **DIM**. The array element subscripts in the result are (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n), where n is the rank of **ARRAY** and DIM is the dimension specified by DIM.

If **DIM** is not specified, the function returns the minimum value of all applicable elements.

If **ARRAY** is of type character, all comparisons are done using the ASCII collating sequence.

If ARRAY is zero-sized or the mask array has all .FALSE. values, then:
- If **ARRAY** is of type integer or real, the result value is the positive number of the largest magnitude, of the same type and kind type as **ARRAY**.
- If **ARRAY** is of type character, each character of the result has the value of CHAR(127).

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:
- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless
- **MASK** if it is a scalar of type logical

Examples

```
! A is the array  | -41  33 25 |
!                 |  12 -61 11 |

! What is the smallest element in A?
      RES = MINVAL(A)
! The result is -61

! What is the smallest element in each column of A?
      RES = MINVAL(A, DIM=1)
! The result is | -41 -61 11 |

! What is the smallest element in each row of A?
      RES = MINVAL(A, DIM=2)
! The result is | -41 -61 |

! What is the smallest element in each row of A,
! considering only those elements that are
! greater than zero?
      RES = MINVAL(A, DIM=2, MASK = A .GT.0)
! The result is | 25 11 |
```

MOD(A, P)
Purpose

Remainder function.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER or REAL

P

An INTENT(IN) argument of the same type and kind type parameter as A.

IBM The kind type parameters can be different if the compiler option –qport=mod is specified. **IBM**

Result type and attributes

Same as A.

Result value

- If P ≠ 0, the value of the result is A - INT(A/P) * P.
- If P = 0, the result is undefined.

Examples

MOD (3.0, 2.0) has the value 1.0.
MOD (8, 5) has the value 3.
MOD (-8, 5) has the value -3.
MOD (8, -5) has the value 3.
MOD (-8, -5) has the value -3.

Specific Name	Argument Type	Result Type	Pass As Arg?
MOD	default integer	default integer	yes
AMOD	default real	default real	yes
DMOD	double precision real	double precision real	yes
QMOD	REAL(16)	REAL(16)	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

Related information

For information on alternative behavior for **MOD** when porting programs to XL Fortran, see the **-qport** compiler option in the *XL Fortran Compiler Reference*.

MODULO(A, P)
Purpose

Modulo function.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER or REAL

P   An `INTENT(IN)` argument of the same type and kind type parameter as A.

Result type and attributes

Same as A.

Result value

- Case (i): A is of type integer. If P ≠ 0, MODULO (A, P) has the value R such that A = Q * P + R, where Q is an integer.

  If P > 0, the inequalities 0 ≤ R < P hold.

  If P < 0, P < R ≤ 0 hold.

  If P = 0, the result is undefined.

- Case (ii): A is of type real. If P ≠ 0, the value of the result is A - FLOOR (A / P) * P.

  If P = 0, the result is undefined.

Examples
```
MODULO (8, 5) has the value 3.
MODULO (-8, 5) has the value 2.
MODULO (8, -5) has the value -2.
MODULO (-8, -5) has the value -3.
```

MOVE_ALLOC(FROM, TO) (Fortran 2003)

Purpose

Allows you to move allocation status, dynamic type, type parameter values, bounds information, and values from one object to another.

Class

subroutine

Argument type and attributes

FROM
      An `INTENT(INOUT)` argument. The corresponding actual argument must be an allocatable object. The actual argument can be a scalar or an array.

TO   An `INTENT(INOUT)` argument. The corresponding actual argument must be an allocatable object. It must be type-compatible and have the same rank as **FROM**. It must be polymorphic if **FROM** is polymorphic. Each nondeferred parameter of the declared type of the actual argument associated with **TO** must have the same value as the corresponding parameter of the declared type of the actual argument associated with **FROM**.

Result value

If **FROM** is deallocated, the allocation status of **TO** is deallocated.

If **FROM** is allocated, **TO** is allocated with the same dynamic type, type parameters, array bounds, and value as those of **FROM**.

If **TO** has the **TARGET** attribute, any pointer associated with **FROM** is correspondingly associated with **TO**.

If **TO** does not have the **TARGET** attribute, the association status of any pointer that was associated with **FROM** when you call **MOVE_ALLOC** becomes undefined.

MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

Purpose

Copies a sequence of bits from one data object to another.

Class

Elemental subroutine

Argument type and attributes

FROM
> An INTENT(IN) INTEGER

FROMPOS
> An INTENT(IN) INTEGER. Its value must be nonnegative. FROMPOS + LEN must be less than or equal to BIT_SIZE (FROM).

LEN An INTENT(IN) INTEGER. Its value must be nonnegative.

TO An INTENT(INOUT) INTEGER variable with the same kind type parameter value as FROM and may be the same variable as FROM. TO is set by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry.

> The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

TOPOS
> An INTENT(IN) INTEGER. Its value must be nonnegative. TOPOS + LEN must be less than or equal to BIT_SIZE (TO).

Examples

If TO has the initial value 6, the value of TO is 5 after the statement
```
CALL MVBITS (7, 2, 2, TO, 0)
```

See "Integer bit model" on page 570.

NEAREST(X,S)

Purpose

Returns the nearest different processor-representable number in the direction indicated by the sign of S (toward positive or negative infinity).

Class

Elemental function

Argument type and attributes

X An INTENT(IN) REAL

S An INTENT(IN) REAL. Its value cannot be zero.

Result type and attributes

Same as X.

Result value

The result is the machine number different from and nearest to X in the direction of the infinity with the same sign as S.

Examples

IBM **NEAREST** (3.0, 2.0) = 3.0 + 2.0$^{(-22)}$. See "Real data model" on page 571. IBM

NEW_LINE(A) (Fortran 2003)
Purpose

The **NEW_LINE** intrinsic returns a new line character.

Class

Inquiry function

Argument type and attributes

A   An INTENT(IN) character. The actual argument corresponding to **A** can be scalar or array.

Result type and attributes

Character scalar of length one.

Result value

The result is the same as ACHAR(10).

Examples

The following example uses the **NEW_LINE** intrinsic in list-directed output:

```
character(1) c
print *, 'The first sentence.', NEW_LINE(c), 'The second sentence.'
```

Expected Output:

```
The first sentence.
The second sentence.
```

The following example passes a character literal constant to the **NEW_LINE** intrinsic:

```
character(100)  line
line = 'IBM' // NEW_LINE('Fortran') // 'XL Fortran Compiler'
```

Expected Output:

```
IBM
XL Fortran Compiler
```

NINT(A, KIND)

Purpose

Nearest integer.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN) REAL`

KIND (optional)
An `INTENT(IN) INTEGER` scalar. The actual argument corresponding to
KIND must be a constant expression. `F2003`

Result type and attributes

- Integer.
- If **KIND** is present, the kind type parameter is that specified by **KIND**;
  otherwise, the kind type parameter is that of the default integer type.

Result value

- If A > 0, NINT (A) has the value INT (A + 0.5).
- If A ≤ 0, NINT (A) has the value INT (A - 0.5).
- The result is undefined if its value cannot be represented in the specified integer
  type.

Examples

NINT (2.789) has the value 3. **NINT** (2.123) has the value 2.

Specific Name	Argument Type	Result Type	Pass As Arg?
NINT	default real	default integer	yes
IDNINT	double precision real	default integer	yes
IQNINT	REAL(16)	default integer	yes **1**
Note: **1** IBM extension			

NOT(I)

Purpose

Performs a bitwise complement of integer.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN) INTEGER`

Result type and attributes

Same as I.

Result value

The result has the value obtained by complementing I bit-by-bit according to the following table:

```
ith bit    ith bit
 of I    of NOT (I)
------------------
  1          0
  0          1
```

The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

If I is represented by the string of bits `01010101`, NOT (I) has the string of bits `10101010`. See "Integer bit model" on page 570.

Specific Name	Argument Type	Result Type	Pass As Arg?
NOT	any integer	same as argument	yes **1**
Note: **1** IBM extension			

NULL(MOLD)
Purpose

This function returns a pointer or designates a deallocated allocatable component of a structure constructor. The association status of the pointer is disassociated.

You must use the function without the **MOLD** argument in any of the following:
- initialization of an object in a declaration
- default initialization of a component
- in a **DATA** statement
- in a **STATIC** statement

You can use the function with or without the **MOLD** argument in any of the following:
- in the **PARAMETER** attribute
- on the right side of a pointer assignment
- in a structure constructor
- as an actual argument

Class

Transformational function.

Argument type and attributes

MOLD (optional)
An `INTENT(IN)` argument. The corresponding actual argument must be a

pointer or allocatable. The actual argument can be of any type or can be a procedure pointer. The association status of the pointer can be undefined, disassociated, or associated. If the actual argument has an association status of associated, the target may be undefined. If the actual argument is allocatable, its allocation status can be allocated or deallocated.

Result type and attributes

If **MOLD** is present, the result's characteristics are the same as those of **MOLD**. If **MOLD** has deferred type parameters, those type parameters of the result are deferred. If **MOLD** is not present, the entity's type, type parameter and rank are determined as follows:

- same as the pointer that appears on the left hand side, for a pointer assignment
- same as the object, when initializing an object in a declaration
- same as the component, in a default initialization for a component
- same as the corresponding component, in a structure constructor
- same as the corresponding dummy argument, as an actual argument
- same as the corresponding pointer object, in a **DATA** statement
- same as the corresponding pointer object, in a **STATIC** statement

Result value

The result is a pointer with disassociated association status or an unallocated allocatable entity.

Examples

```
! Using NULL() as an actual argument.
INTERFACE
   SUBROUTINE FOO(I, PR)
      INTEGER I
      REAL, POINTER:: PR
   END SUBROUTINE FOO
END INTERFACE

CALL FOO(5, NULL())
```

NUM_PARTHDS() (IBM extension)
Purpose

Note: This intrinsic procedure is deprecated and might be removed in a future release.

Returns the number of parallel Fortran threads the run time should create during execution of a program. This value is set by using the **PARTHDS** runtime option. If the user does not set the **PARTHDS** runtime option, the run time will set a default value for **PARTHDS**. In doing so, the run time may consider the following aspects when setting the option:

- The number of processors on the machine
- The value specified in the run-time option **USRTHDS**

Class

Inquiry function

Result value

Default scalar integer

If the compiler option **-qsmp** has not been specified, then **NUM_PARTHDS** will always return a value of 1.

Examples

```
I = NUM_PARTHDS()
IF (I == 1) THEN
   CALL SINGLE_THREAD_ROUTINE()
ELSE
   CALL MULTI_THREAD_ROUTINE()
```

Specific Name	Result Type	Pass As Arg?
NUM_PARTHDS	default scalar integer	no

Related information

See the **parthds** and **XLSMPOPTS** runtime options in the *XL Fortran Optimization and Programming Guide*.

NUM_USRTHDS() (IBM extension)
Purpose

Note: This intrinsic procedure is deprecated and might be removed in a future release.

Returns the number of threads that will be explicitly created by the user during execution of the program. This value is set by using the **USRTHDS** runtime option.

Class

Inquiry function

Result value

Default scalar integer

If the value has not been explicitly set using the **USRTHDS** runtime option, the default value is 0.

Specific Name	Result Type	Pass As Arg?
NUM_USRTHDS	default scalar integer	no

Related information

See the **usrthds** and the **XLSMPOPTS** runtime options in the *XL Fortran Optimization and Programming Guide*.

NUMBER_OF_PROCESSORS(DIM) (IBM extension)
Purpose

Returns a scalar of type default integer whose value is always 1. This intrinsic ensures compatibility with programs written for High Performance Fortran (HPF) environments.

Class

System inquiry function

Argument type and attributes

DIM (optional)
An `INTENT(IN) INTEGER` scalar that has a value of 1 (the rank of the processor array).

Result type and attributes

Default scalar integer which always has a value of 1.

Examples

```
I = NUMBER_OF_PROCESSORS()       ! 1
J = NUMBER_OF_PROCESSORS(DIM=1)  ! 1
```

PACK(ARRAY, MASK, VECTOR)
Purpose

Takes some or all elements from an array and packs them into a one-dimensional array, under the control of a mask.

Class

Transformational function

Argument type and attributes

ARRAY
The source array, whose elements become part of the result. The corresponding actual argument can have any data type.

MASK
An argument of type `LOGICAL` that conforms with **ARRAY**. It determines which elements are taken from the source array. If it is a scalar, its value applies to all elements in **ARRAY**.

VECTOR (optional)
A padding array whose elements are used to fill out the result if there are not enough elements selected by the mask. It is a one-dimensional array that has the same data type and type parameters as **ARRAY** and at least as many elements as there are true values in **MASK**. If **MASK** is a scalar with a value of .TRUE., **VECTOR** must have at least as many elements as there are array elements in **ARRAY**.

Result value

The result is always a one-dimensional array with the same data type and type parameters as ARRAY.

The size of the result depends on the optional arguments:
- If VECTOR is specified, the size of the resultant array equals the size of VECTOR.
- Otherwise, it equals the number of true array elements in MASK, or the number of elements in ARRAY if MASK is a scalar with a value of .TRUE..

The array elements in ARRAY are taken in array element order to form the result. If the corresponding array element in MASK is .TRUE., the element from ARRAY is placed at the end of the result.

If any elements remain empty in the result (because VECTOR is present, and has more elements than there are .TRUE. values in mask), the remaining elements in the result are set to the corresponding values from VECTOR.

Examples

```
! A is the array  | 0  7  0 |
!                 | 1  0  3 |
!                 | 4  0  0 |

! Take only the non-zero elements of this sparse array.
! If there are less than six, fill in -1 for the rest.
RES = PACK(A, MASK= A .NE. 0, VECTOR=(/-1,-1,-1,-1,-1,-1/)
! The result is (/ 1, 4, 7, 3, -1, -1 /).

! Elements 1, 4, 7, and 3 are taken in order from A
! because the value of MASK is true only for these
! elements. The -1s are added to the result from VECTOR
! because the length (6) of VECTOR exceeds the number
! of .TRUE. values (4) in MASK.
```

POPCNT(I) (Fortran 2008)
Purpose

Population count

Counts the number of set bits in a data object.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

> **IBM** The argument can also be of type BYTE, LOGICAL, or REAL. If the type of the argument is REAL, it must not be REAL(16). **IBM**

Result type and attributes

Default integer

Result value

The number of bits set to 1 in the sequence of bits of **I**

Examples

The following table shows the functionality of the POPCNT function.

Integer	Bit Representation	POPCNT
0	0000	0
1	0001	1
2	0010	1
3	0011	2
4	0100	1

Related information

Data representation models

POPPAR(I) (Fortran 2008)
Purpose

Population parity

Determines the parity for a data object.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN) INTEGER`

> `IBM` The argument can also be of type `BYTE`, `LOGICAL`, or `REAL`. If the type of the argument is `REAL`, it must not be `REAL(16)`. `IBM`

Result type and attributes

Default integer

Result value
- Returns 1 if **I** includes an odd number of bits set to 1.
- Returns 0 if **I** includes an even number of bits set to 1.

Examples

The following table shows the functionality of the POPPAR function.

Integer	Bit Representation	POPPAR
0	0000	0
1	0001	1

2	0010	1
3	0011	0
4	0100	1

Related information

Data representation models

PRECISION(X)
Purpose

Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An INTENT(IN) REAL or COMPLEX. The actual argument corresponding to **X** can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

The result is:
```
INT( (DIGITS(X) - 1) * LOG10(2) )
```

IBM Therefore,
```
     Type                Precision
-------------------      ---------
 real(4) , complex(4)         6
 real(8) , complex(8)        15
real(16) , complex(16)       31
```

IBM

Examples

IBM **PRECISION** (X) = INT((24 - 1) * LOG10(2.)) = INT(6.92 ...) = 6 for X of type real(4). See "Real data model" on page 571. **IBM**

PRESENT(A)
Purpose

Determine whether an optional argument is present. If it is not present, you may only pass it as an optional argument to another procedure or pass it as an argument to PRESENT.

Class

Inquiry function

Argument type and attributes

A   An INTENT(IN) argument. The actual argument corresponding to **A** is an optional dummy argument that is accessible in the procedure in which the **PRESENT** function reference appears.

Result type and attributes

Default logical scalar.

Result value

The result is .TRUE. if the actual argument is present (that is, if it was passed to the current procedure in the specified dummy argument), and .FALSE. otherwise.

Examples

```
      SUBROUTINE SUB (X, Y)
        REAL, OPTIONAL :: Y
        IF (PRESENT (Y)) THEN
! In this section, we can use y like any other variable.
          X = X + Y
          PRINT *, SQRT(Y)
        ELSE
! In this section, we cannot define or reference y.
          X = X + 5
! We can pass it to another procedure, but only if
! sub2 declares the corresponding argument as optional.
          CALL SUB2 (Z, Y)
        ENDIF
      END SUBROUTINE SUB
```

Related information

"OPTIONAL" on page 423

PROCESSORS_SHAPE() (IBM extension)
Purpose

Returns a zero-sized array. This intrinsic ensures compatibility with programs written for High Performance Fortran (HPF) environments.

Class

System inquiry function

Result type and attributes

Default integer array of rank one, whose size is equal to the rank of the processor array. In a uniprocessor environment, the result is a zero-sized vector.

Result value

The value of the result is the shape of the processor array.

Examples

```
I=PROCESSORS_SHAPE()
! Zero-sized vector of type default integer
```

PRODUCT(ARRAY, DIM, MASK) or PRODUCT(ARRAY, MASK)
Purpose

Multiplies together all elements in an entire array, or selected elements from all vectors in a specified dimension of an array.

Class

Transformational function

Argument type and attributes

ARRAY
> An array with a numeric data type.

DIM (optional)
> An `INTEGER` scalar (a specified dimension of ARRAY). Its value must be in the range $1 \le \text{DIM} \le \text{rank(ARRAY)}$.

MASK (optional)
> A `LOGICAL` expression that conforms with ARRAY in shape. If MASK is a scalar, the scalar value applies to all elements in ARRAY.

Result value

If **DIM** is present, the result is an array of rank rank(ARRAY)-1 and the same data type and kind type parameter as ARRAY. If DIM is missing, or if MASK has a rank of one, the result is a scalar.

The result is calculated by one of the following methods:

Method 1:
> If only ARRAY is specified, the result is the product of all its array elements. If ARRAY is a zero-sized array, the result is equal to one.

Method 2:
> If ARRAY and MASK are both specified, the result is the product of those array elements of ARRAY that have a corresponding true array element in MASK. If MASK has no elements with a value of .TRUE., the result is equal to one.

Method 3:
> If DIM is also specified and ARRAY has a rank of one, the result is a scalar equal to the product of all elements of ARRAY that have a corresponding .TRUE. array element in MASK.
>
> If DIM is also specified and ARRAY has rank greater than one, the result is a new array in which dimension DIM has been eliminated. Each new array element is the product of elements from a corresponding vector within ARRAY. The index values of that vector, in all dimensions except DIM, match those of the output element. The output element is the product of those vector elements that have a corresponding .TRUE. array element in MASK.

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless
- **MASK** if it is a scalar of type logical

Examples

- Method 1:

```
! Multiply all elements in an array.
       RES = PRODUCT( (/2, 3, 4/) )
! The result is 24 because (2 * 3 * 4) = 24.

! Do the same for a two-dimensional array A, where
! A is the array | 2  3  4 |
!                | 4  5  6 |
       RES = PRODUCT(A)
! The result is 2880.  All elements are multiplied.
```

- Method 2:

```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Multiply all elements of the array that are > -5.
       RES = PRODUCT(A, MASK = A .GT. -5)
! The result is -18 because (-3 * 2 * 3) = -18.
```

- Method 3:

```
! A is the array | -2  5  7 |
!                |  3 -4  3 |
! Find the product of each column in A.
       RES = PRODUCT(A, DIM = 1)
! The result is | -6 -20 21 | because (-2 * 3) =  -6
!                                     ( 5 * -4 ) = -20
!                                     ( 7 *  3 ) = 21

! Find the product of each row in A.
       RES = PRODUCT(A, DIM = 2)
! The result is | -70 -36 |
! because (-2 *  5 * 7) = -70
!         ( 3 * -4 * 3) = -36

! Find the product of each row in A, considering
! only those elements greater than zero.
       RES = PRODUCT(A, DIM = 2, MASK = A .GT. 0)
! The result is | 35 9 | because ( 5 * 7) = 35
!                               ( 3 * 3) =  9
```

QCMPLX(X, Y) (IBM extension)
Purpose

Convert to extended complex type.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER, REAL, or COMPLEX

Y (optional)
        An `INTENT(IN)` `INTEGER` or `REAL`. It must not be present if X is of type `COMPLEX`.

Result type and attributes

It is of type extended complex.

Result value

- If Y is absent and X is not complex, it is as if Y were present with the value of zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X) and X were present with the value REAL(X).
- QCMPLX(X, Y) has the complex value whose real part is REAL(X, KIND=16) and whose imaginary part is REAL(Y, KIND=16).

Examples

QCMPLX (-3) has the value (-3.0Q0, 0.0Q0).

Specific Name	Argument Type	Result Type	Pass As Arg?
QCMPLX	REAL(16)	COMPLEX(16)	no

Related information

"CMPLX(X, Y, KIND)" on page 593, "DCMPLX(X, Y) (IBM extension)" on page 605.

QEXT(A) (IBM extension)
Purpose

Convert to extended precision real type.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN)` `INTEGER` or `REAL`

Result type and attributes

Extended precision real.

Result value

- If A is of type extended precision real, QEXT(A) = A.
- If A is of type integer or real, the result is the exact extended precision representation of A.

Examples

QEXT (-3) has the value -3.0Q0.

Specific Name	Argument Type	Result Type	Pass As Arg?
QFLOAT	any integer	REAL(16)	no
QEXT	default real	REAL(16)	no
QEXTD	double precision real	REAL(16)	no

RADIX(X)
Purpose

Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An `INTENT(IN)` `INTEGER` or `REAL`. The actual argument corresponding to **X** can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

The result is the base of the model representing numbers of the same kind and type as X. IBM The result is always 2. IBM See the models under "Data representation models" on page 570.

RAND() (IBM extension)
Purpose

Not recommended. Generates uniform random numbers, positive real numbers greater than or equal to 0.0 and less than 1.0. Instead, use the standards conforming **RANDOM_NUMBER(HARVEST)** intrinsic subroutine.

Class

None (does not correspond to any of the defined categories).

Result type and attributes

A `REAL(4)` scalar

Related information

"SRAND(SEED) (IBM extension)" on page 710 can be used to specify a seed value for the random number sequence.

If the function result is assigned to an array, all array elements receive the same value.

Examples

See the following example of a program using the **RAND** function.

```
DO I = 1, 5
   R = RAND()
   PRINT *, R
ENDDO
END
```

See the sample output generated by the above program:

```
0.2251586914
0.8285522461
0.6456298828
0.2496948242
0.2215576172
```

This function only has a specific name.

RANDOM_NUMBER(HARVEST)
Purpose

Returns one pseudo-random number or an array of pseudo-random numbers from the uniform distribution over the range $0 \leq x < 1$.

If you link the **libpthreads.a** library, a parallel implementation of random number generation is employed which improves performance on SMP machines. The number of threads used can be controlled by the **intrinthds**=*num* run-time option.

Class

Subroutine

Argument type and attributes

HARVEST
> An INTENT(OUT) REAL. The actual argument corresponding to **HARVEST** can be a scalar or an array variable. It is set to pseudo-random numbers from the uniform distribution in the interval $0 \leq x < 1$.

Examples

```
REAL X, Y (10, 10)
! Initialize X with a pseudo-random number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X and Y contain uniformly distributed random numbers
```

RANDOM_SEED(SIZE, PUT, GET, GENERATOR)
Purpose

Restarts or queries the pseudo-random number generator used by RANDOM_NUMBER.

Class

Subroutine

Argument type and attributes

There must either be exactly one or no arguments present.

SIZE (optional)
> An `INTENT(OUT)` `INTEGER` scalar. It is set to the number of default type integers (N) that are needed to hold the value of the seed, which is an 8-byte variable.

PUT (optional)
> An `INTENT(IN)` `INTEGER` array of rank one and size ≥ N. The seed for the current generator is transferred from it.

GET (optional)
> An `INTENT(OUT)` `INTEGER` array of rank one and size ≥ N. The seed for the current generator is transferred to it.

IBM **GENERATOR (optional)**
> An `INTENT(IN)` `INTEGER` scalar. Its value determines the random number generator to be used subsequently. The value must be either 1 or 2. **IBM**

IBM Random_seed allows the user to toggle between two random number generators. Generator 1 is the default. Each generator maintains a private seed and normally resumes its cycle after the last number it generated. A valid seed must be a whole number in the range 1.0 to 2147483647.0 (2.0**31-1) for Generator 1 and in the range 1.0 to 281474976710656.0 (2.0**48) for Generator 2.

Generator 1 uses the multiplicative congruential method, with

```
S(I+1) = ( 16807.0 * S(I) ) mod (2.0**31-1)
```

and

```
X(I+1) = S(I+1) / (2.0**31-1)
```

Generator 1 cycles after 2**31-2 random numbers.

Generator 2 also uses the multiplicative congruential method, with

```
S(I+1) = ( 44,485,709,377,909.0 * S(I) )
              mod (2.0**48)
```

and

```
X(I+1) = S(I+1) / (2.0**48)
```

Generator 2 cycles after (2**48) random numbers. Although generator 1 is the default (for reasons of backwards compatibility) the use of generator 2 is recommended for new programs since it typically runs faster than generator 1 and has a longer period.

If no argument is present, the seed of the current generator is set to the default value 1d0. **IBM**

Examples

```
CALL RANDOM_SEED
   ! Current generator sets its seed to 1d0
CALL RANDOM_SEED (SIZE = K)
   ! Sets K = 64 / BIT_SIZE( 0 )
```

```
CALL RANDOM_SEED (PUT = SEED (1 : K))
    ! Transfer seed to current generator
CALL RANDOM_SEED (GET = OLD (1 : K))
    ! Transfer seed from current generator
```

RANGE(X)

Purpose

Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X   An INTENT(IN) INTEGER, REAL, or COMPLEX. The actual argument corresponding to **X** can be scalar or array valued.

Result type and attributes

Default integer scalar.

Result value

1. For an integer argument, the result is:

   INT(LOG10(HUGE(X)))

2. For a real or complex argument, the result is:

   INT(MIN(LOG10(HUGE(X)), -LOG10(TINY(X))))

IBM▶ Thus:

```
Type                       RANGE
-----------------------------
integer(1)                     2
integer(2)                     4
integer(4)                     9
integer(8)                    18
real(4) , complex(4)          37
real(8) , complex(8)         307
real(16) , complex(16)       291
```

◀IBM

Examples

IBM▶ **X** is of type real(4):
     HUGE(X) = 0.34E+39
     TINY(X) = 0.11E-37
     RANGE(X) = 37

◀IBM

See "Data representation models" on page 570.

RANK(A) (TS 29113)
Purpose

Returns the rank of a data object, such as an assumed-rank object.

Class

Inquiry function

Argument type and attributes

A   An **INTENT(IN)** argument. The corresponding actual argument can be an array or a scalar of any type.

Result type and attributes

Default integer scalar.

Result value

The result is the rank of **A**.

Examples
```
REAL :: a(10, 20)
PRINT *, RANK(a)
END
```

The output is as follows:
```
2
```

REAL(A, KIND)
Purpose

Convert to real type.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER, REAL, COMPLEX or boz-literal CONSTANT

KIND (optional)
An INTENT(IN) INTEGER scalar. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- Real.
- Case (i): If A is of type integer or real and **KIND** is present, the kind type parameter is that specified by **KIND**. If A is of type integer or real and **KIND** is not present, the kind type parameter is the kind type parameter of the default real type.
- Case (ii): If A is of type complex and **KIND** is present, the kind type parameter is that specified by **KIND**. If A is of type complex and **KIND** is not present, the kind type parameter is the kind type parameter of A.

- Case (iii): If A is a *boz-literal* constant and **KIND** is present, the kind type parameter is that specified by **KIND**. If A is a *boz-literal* constant and **KIND** is not present, the kind type parameter is that of default real type. If `-qxlf2003=nobozlitargs` is specified the boz-literal constant is treated as an integer.

Result value

- Case (i): If A is of type integer or real, the result is equal to a kind-dependent approximation to A.
- Case (ii): If A is of type complex, the result is equal to a kind-dependent approximation to the real part of A.
- Case (iii): If A is a *boz-literal* constant, the value of the result is equal to the value that a variable of the same type and kind type parameters as the result would have if its value were the bit pattern specified by the *boz-literal* constant.

Examples

REAL (-3) has the value -3.0. **REAL** ((3.2, 2.1)) has the value 3.2.

Specific Name	Argument Type	Result Type	Pass As Arg?
REAL	default integer	default real	no
FLOAT	any integer **1**	default real	no
SNGL	double precision real	default real	no
SNGLQ	REAL(16)	default real	no **2**
DREAL	double complex	double precision real	no **2**
QREAL	COMPLEX(16)	REAL(16)	no **2**
Note: **1** IBM extension: the ability to specify a nondefault integer argument. **2** IBM extension: the inability to pass the name as an argument.			

F2008 In Fortran 2008, you can use *designator*%RE to access the real part of complex numbers directly; for instance, A%RE has the same value as REAL(A). For more information about complex part designators, see Complex. F2008

REPEAT(STRING, NCOPIES)

Purpose

Concatenate several copies of a string.

Class

Transformational function

Argument type and attributes

STRING
>An INTENT(IN) CHARACTER scalar.

NCOPIES
>An INTENT(IN) INTEGER scalar. Its value must not be negative.

Result type and attributes

Character scalar with a length equal to NCOPIES * LENGTH(STRING), with the same kind type parameter as STRING.

Result value

The value of the result is the concatenation of NCOPIES copies of STRING.

Examples

REPEAT ('H', 2) has the value 'HH'. **REPEAT** ('XYZ', 0) has the value of a zero-length string.

RESHAPE(SOURCE, SHAPE, PAD, ORDER)
Purpose

Constructs an array of a specified shape from the elements of a given array.

Class

Transformational function

Argument type and attributes

SOURCE
> An array of any type, which supplies the elements for the result array.

SHAPE
> An `INTEGER` array of up to 20 elements, with rank one and of a constant size. It defines the shape of the result array. All elements are either positive integers or zero.

PAD (optional)
> An array of the same data type and type parameters as SOURCE. It is used to fill in extra values if SOURCE is reshaped into a larger array. If it is absent or is a zero-sized array, you can only make SOURCE into another array of the same size or smaller.

ORDER (optional)
> An `INTEGER` array of rank one with a constant size. Its elements must be a permutation of (1, 2, ..., SIZE(**SHAPE**)). You can use it to insert elements in the result in an order of dimensions other than the normal (1, 2, ..., rank(RESULT)).

Result value

The result is an array with shape SHAPE. It has the same data type and type parameters as SOURCE.

The array elements of SOURCE are placed into the result in the order of dimensions as specified by ORDER, or in the usual order for array elements if ORDER is not specified.

The array elements of SOURCE are followed by the array elements of PAD in array element order, and followed by additional copies of PAD until all of the elements of the result are set.

Examples

```
! Turn a rank-1 array into a 3x4 array of the
! same size.
RES= RESHAPE( (/A,B,C,D,E,F,G,H,I,J,K,L/), (/3,4/)
! The result is │ A D G J │
!               │ B E H K │
!               │ C F I L │

! Turn a rank-1 array into a larger 3x5 array.
! Keep repeating -1 and -2 values for any
! elements not filled by the source array.
! Fill the rows first, then the columns.
RES= RESHAPE( (/1,2,3,4,5,6/), (/3,5/), &
  (/-1,-2/), (/2,1/) )
! The result is │ 1  2  3  4  5 │
!               │    6 -1 -2 -1 -2 │
!               │ -1 -2 -1 -2 -1 │
```

Related information

"SHAPE(SOURCE, KIND)" on page 698.

RRSPACING(X)
Purpose

Returns the reciprocal of the relative spacing of the model numbers near the argument value.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

The result is:

```
ABS(FRACTION(X)) * FLOAT(RADIX(X))^DIGITS(X)
```

Examples

IBM **RRSPACING** $(-3.0) = 0.75 * 2^{24}$. See "Real data model" on page 571. IBM

RSHIFT(I, SHIFT) (IBM extension)
Purpose

Performs an arithmetic shift to the right. It is identical to "SHIFTA(I, SHIFT) (Fortran 2008)" on page 699.

Note: You are encouraged to use the standard intrinsic procedure rather than the equivalent IBM extension.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` INTEGER

SHIFT
>   An `INTENT(IN)` INTEGER. Its value must be non-negative and less than or equal to BIT_SIZE(I).

Result type and attributes

Same as I.

Result value
- The result has the value obtained by shifting the bits of I by SHIFT positions to the right.
- Vacated bits are filled with the sign bit.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Examples

RSHIFT (3, 1) has the result 1.

RSHIFT (3, 2) has the result 0.

RSHIFT (-1, 32) has the result -1.

Specific Name	Argument Type	Result Type	Pass As Arg?
RSHIFT	any integer	same as argument	yes

SAME_TYPE_AS(A,B) (Fortran 2003)
Purpose

Inquires whether the dynamic type of A is the same as the dynamic type of B.

Class

Inquiry function

Argument type and attributes

A   An `INTENT(IN)` argument. The corresponding actual argument must be of extensible type. If the actual argument is a pointer, it must not have an undefined association status.

B   An `INTENT(IN)` argument. The corresponding actual argument must be of extensible type. If the actual argument is a pointer, it must not have an undefined association status.

Result type and attributes

Default logical scalar

Result value

The result is true if the dynamic type of **A** is the same as the dynamic type of **B**.

Note: The result depends only on the dynamic types of **A** and **B**. Differences in type parameters are ignored.

Related information
- "Extensible derived types (Fortran 2003)" on page 58

SCALE(X,I)
Purpose

Returns the scaled value: $X * 2.0^I$

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL

I   An `INTENT(IN)` INTEGER

Result type and attributes

Same as X.

Result value

IBM▶ The result is determined from the following value:

$X * 2.0^I$ ◀IBM

Examples

IBM▶ **SCALE** $(4.0, 3) = 4.0 * (2^3) = 32.0$. See "Real data model" on page 571. ◀IBM

SCAN(STRING, SET, BACK, KIND)
Purpose

Scan a string for any one of the characters in a set of characters.

Class

Elemental function

Argument type and attributes

STRING
        An `INTENT(IN)` character.

SET   An `INTENT(IN)` character with the same kind type parameter as **STRING**.

BACK (optional)
        An `INTENT(IN)` logical.

F2003 **KIND (optional)** **F2003**

An INTENT(IN) integer. The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- An integer.
- **F2003** If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. **F2003**

Result value

- Case (i): If BACK is absent or is present with the value .FALSE. and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.
- Case (ii): If BACK is present with the value .TRUE. and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.
- Case (iii): The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

Examples

- Case (i): SCAN ('FORTRAN', 'TR') has the value 3.
- Case (ii): SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.
- Case (iii): SCAN ('FORTRAN', 'BCD') has the value 0.

SELECTED_CHAR_KIND(NAME) (Fortran 2003)

Purpose

Returns a value of the kind type parameter of a character data type.

Class

Transformational function

Argument type and attributes

NAME
An INTENT(IN) CHARACTER scalar

Result type and attributes

Default integer scalar.

Result value

- If you specify **NAME** as ASCII, **SELECTED_CHAR_KIND** returns the kind type parameter of the ASCII character type.
- If you specify **NAME** as DEFAULT, **SELECTED_CHAR_KIND** returns the kind type parameter of the default character type
- Otherwise, **SELECTED_CHAR_KIND** returns -1.

Related information

Kind type parameters supported by XL Fortran are defined in "Type declaration: type parameters and specifiers" on page 17.

SELECTED_INT_KIND(R)
Purpose

Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$.

Class

Transformational function

Argument type and attributes

R   An INTENT(IN) INTEGER

Result type and attributes

Default integer scalar.

Result value

- The result has a value equal to the value of the kind type parameter of an integer data type that represents all values n in the range values n with $-10^R < n < 10^R$, or if no such kind type parameter is available, the result is -1.
- If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range.

Examples

IBM **SELECTED_INT_KIND** (9) has the value 4, signifying that an INTEGER with kind type 4 can represent all values from 10^{-9} to 10^9. IBM

Related information

Kind type parameters supported by XL Fortran are defined in "Type declaration: type parameters and specifiers" on page 17.

SELECTED_REAL_KIND(P, R, RADIX)
Purpose

Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits, a decimal exponent range of at least R, F2008 and a radix of RADIX. F2008

Class

Transformational function

Argument type and attributes

At least one argument must be present.

P (optional)
An `INTENT(IN)` `INTEGER` scalar

R (optional)
An `INTENT(IN)` `INTEGER` scalar

`F2008` **RADIX (optional)**
An `INTENT(IN)` `INTEGER` scalar `F2008`

Result type and attributes

Default integer scalar.

Result value

If P or R is not specified, **SELECTED_REAL_KIND** behaves as if you specified P or R with value 0. If RADIX is not specified, the radix of the selected kind can be any supported value.

The result is the value of the kind type parameter of a real data type that satisfies the following conditions:

- It has decimal precision, as returned by the **PRECISION** function, of at least P digits.
- It has a decimal exponent range, as returned by the**RANGE** function, of at least R.
- `F2008` It has a radix, as returned by the **RADIX** function, of RADIX. `F2008`

If no such kind type parameter is available, the result has different values depending on different conditions as follows:

- If `F2008` the radix is available `F2008`, the precision is not available, and the exponent range is available, the result is -1.
- If `F2008` the radix is available `F2008`, the exponent range is not available, and the precision is available, the result is -2.
- If `F2008` the radix is available `F2008`, and neither the precision nor the exponent range is available, the result is -3.
- If `F2008` the radix is available `F2008`, and both the precision and exponent range are available separately but not together, the result is -4.
- `F2008` If the radix is not available, the result is -5. `F2008`

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. However, if several values have the same smallest decimal precision, the smallest value is returned.

`F2008` **Currently, the XL Fortran compiler only supports** *RADIX*=2. `F2008`

Examples

The following example shows the usage of the **SELECTED_REAL_KIND** intrinsic procedure.

```
PROGRAM a
  INTEGER :: i

  i = SELECTED_REAL_KIND(6, 70)
  PRINT *, 'SELECTREALKIND(6, 70) = ', i
END PROGRAM a
```

The output of this program is as follows:

```
SELECTREALKIND(6, 70) = 8
```

SELECTED_REAL_KIND (6, 70) has the value 8.

F2008

The following example shows the usage of the **SELECTED_REAL_KIND** intrinsic procedure with the RADIX argument.

```
PROGRAM a
  INTEGER :: i

  i = SELECTED_REAL_KIND(20, 140, 2)
  PRINT *, 'SELECTREALKIND(20, 140, 2) = ', i
END PROGRAM a
```

The output of this program is as follows:

```
SELECTREALKIND(20, 140, 2) = 16
```

F2008

Related information
- PRECISION(X)
- RANGE(X)
- "RADIX(X)" on page 684
- Kind type parameters supported by XL Fortran are defined in "Type declaration: type parameters and specifiers" on page 17.

SET_EXPONENT(X,I)
Purpose

Returns the number whose fractional part is the fractional part of the model representation of X, and whose exponent part is I.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

I   An INTENT(IN) INTEGER

Result type and attributes

Same as X.

Result value

──────────────── **IBM extension** ────────────────

If X = 0 the result is zero.

Otherwise, the result is:

```
FRACTION(X) * 2.0^I
```

Example

SET_EXPONENT (10.5, 1) $= 0.65625 \, * \, 2.0^1 \, = \, 1.3125$

See "Real data model" on page 571.

SHAPE(SOURCE, KIND)

Purpose

Returns the shape of an array or scalar.

Class

Inquiry function

Argument type and attributes

SOURCE
An `INTENT(IN)` argument. The corresponding actual argument can be an array or scalar of any data type. The actual argument must not be a disassociated pointer, allocatable object that is not allocated, or assumed-size array.

F2003 **KIND (optional)**
An `INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. F2003

Result type and attributes

- The result is an array of rank one whose size is **RANK**(**SOURCE**).
- F2003 It is of type integer
- If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type. F2003

Result value

The extent of each dimension in **SOURCE** is returned in the corresponding element of the result array.

TS 29113 If **ARRAY** is an assumed-rank object that is associated with an assumed-size array, the result has a value equal to [(**SIZE**(**ARRAY**, **I**, **KIND**), **I**=1, **RANK**(**ARRAY**))]. If **KIND** is omitted from **SHAPE**, **KIND** is also omitted from **SIZE**. TS 29113

Related information

"RESHAPE(SOURCE, SHAPE, PAD, ORDER)" on page 690.

Examples

```
! A is the array │ 7 6 3 1 │
!                │ 2 4 0 9 │
!                │ 5 7 6 8 │
!
        RES = SHAPE( A )
! The result is │ 3  4 │ because A is a rank-2 array
! with 3 elements in each column and 4 elements in
! each row.
```

SHIFTA(I, SHIFT) (Fortran 2008)
Purpose

Performs an arithmetic shift to the right. It is identical to "RSHIFT(I, SHIFT) (IBM extension)" on page 691.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

SHIFT
         An INTENT(IN) INTEGER. Its value must be non-negative, and less than or equal to BIT_SIZE(I).

Result type and attributes

Same as I.

Result value
- The result has the value that is obtained by shifting the bits of I by SHIFT positions to the right.
- Vacated bits are filled with the sign bit, leftmost bit of I.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Example

SHIFTA (-68, 4) has the result -4.

SHIFTL(I, SHIFT) (Fortran 2008)
Purpose

Performs a shift to the left. It is identical to "LSHIFT(I, SHIFT) (IBM extension)" on page 652.

Class

Elemental function

Argument type and attributes

I   An INTENT(IN) INTEGER

SHIFT
> An `INTENT(IN)` `INTEGER`. Its value must be non-negative, and less than or equal to BIT_SIZE(I).

Result type and attributes

Same as I.

Result value
- The result has the value that is obtained by shifting the bits of I by SHIFT positions to the left, the same as ISHFT(I, SHIFT).
- Vacated bits are filled with zero.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Example

SHIFTL (-8, 2) has the result -32.

SHIFTR(I, SHIFT) (Fortran 2008)
Purpose

Performs a logical shift to the right.

Class

Elemental function

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`

SHIFT
> An `INTENT(IN)` `INTEGER`. Its value must be non-negative, and less than or equal to BIT_SIZE(I).

Result type and attributes

Same as I.

Result value
- The result has the value that is obtained by shifting the bits of I by SHIFT positions to the right, the same as "ISHFT(I, SHIFT)" on page 640.
- Vacated bits are filled with zero.
- The bits are numbered 0 to BIT_SIZE(I)-1, from right to left.

Example

SHIFTR (12, 1) has the result 6.

SIGN(A, B)
Purpose

Returns the absolute value of A times the sign of B. If A is non-zero, you can use the result to determine whether B is negative or non-negative, as the sign of the result is the same as the sign of B.

Note that if you have declared B as **REAL(4)** or **REAL(8)**, and B has a negative zero value, the sign of the result depends on whether you have specified the **-qxlf90=signedzero** compiler option.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) INTEGER or REAL

B   An argument of the same type and kind type parameter as A.

Result type and attributes

Same as A.

Result value

The result is $sgn*|A|$, where:
- sgn = -1, if either of the following is true:
  - B < 0
  - **IBM** B is a **REAL(4)** or **REAL(8)** number with a value of negative 0, and you have specified the **-qxlf90=signedzero** option **IBM**
- sgn = 1, otherwise.

Fortran 95 allows a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Using the **-qxlf90=signedzero** option allows you to specify the Fortran 95 behavior (except in the case of **REAL(16)** numbers), which is consistent with the IEEE standard for binary floating-point arithmetic. **-qxlf90=signedzero** is the default for the **xlf95**, **xlf95_r**, **f2003**, and **f2008** invocation commands.

Examples

SIGN (-3.0, 2.0) has the value 3.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SIGN	default real	default real	yes
ISIGN	any integer **1**	same as argument	yes
DSIGN	double precision real	double precision real	yes
QSIGN	REAL(16)	REAL(16)	yes **2**
Note: **1** IBM extension: the ability to specify a nondefault integer argument. **2** IBM extension: the ability to pass the name as an argument.			

Related information

See **-qxlf90** in the *XL Fortran Compiler Reference*.

SIGNAL(I, PROC) (IBM extension)
Purpose

The SIGNAL procedure allows a program to specify a procedure to be invoked upon receipt of a specific operating-system signal.

Class

Subroutine

Argument type and attributes

I   An `INTENT(IN)` `INTEGER`. It specifies the value of the signal to be acted upon. Available signal values are defined in the C include file **signal.h**; a subset of signal values is defined in the Fortran include file **fexcp.h**.

PROC An `INTENT(IN)` argument that specifies the user-defined procedure to be invoked when the process receives the specified signal specified by argument I.

Examples

```
      INCLUDE 'fexcp.h'
      INTEGER   SIGUSR1
      EXTERNAL  USRINT
! Set exception handler to produce the traceback code.
! The SIGTRAP is defined in the include file fexcp.h.
! xl__trce is a procedure in the XL Fortran
! run-time library.  It generates the traceback code.
      CALL SIGNAL(SIGTRAP, XL__TRCE)
      ...
! Use user-defined procedure USRINT to handle the signal
! SIGUSR1.
      CALL SIGNAL(SIGUSR1, USRINT)
      ...
```

Related information

The **-qsigtrap** option in the *XL Fortran Compiler Reference* allows you to set a handler for **SIGTRAP** signals through a compiler option.

SIN(X)
Purpose

Sine function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` `REAL` or `COMPLEX`. If X is `REAL`, it is regarded as a value in radians. If X is `COMPLEX`, its real and imaginary parts are regarded as values in radians.

Result type and attributes

Same as X.

Result value

It approximates sin(X).

Examples

SIN (1.0) has the value 0.84147098 (approximately).

Specific Name	Argument Type	Result Type	Pass As Arg?
SIN	default real	default real	yes
DSIN	double precision real	double precision real	yes
QSIN	REAL(16)	REAL(16)	yes **1**
CSIN **2a**	default complex	default complex	yes
CDSIN **2b**	double complex	double complex	yes **1**
ZSIN **2b**	double complex	double complex	yes **1**
CQSIN **2b**	COMPLEX(16)	COMPLEX(16)	yes **1**
Notes:			

1 IBM extension: the ability to pass the name as an argument.
Given that X is a complex number in the form a + bi, where i = $(-1)^{\frac{1}{2}}$:
2a abs(b) must be less than or equal to 88.7228; a is any real value.
2b abs(b) must be less than or equal to 709.7827; a is any real value.

SIND(X) (IBM extension)
Purpose

Sine function. Argument in degrees.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL

Result type and attributes

Same as X.

Result value

It approximates sin(X), where X has a value in degrees.

Examples

SIND (90.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SIND	default real	default real	yes
DSIND	double precision real	double precision real	yes
QSIND	REAL(16)	REAL(16)	yes

SINH(X)

Purpose

Hyperbolic sine function.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN)` REAL `F2008` or type complex. `F2008`

Result type and attributes

Same as X.

Result value

The result value approximates sinh(X).

`F2008` If X is of type complex, its imaginary part is considered a value in radians. `F2008`

Examples

SINH(1.0) has the value 1.1752012, approximately.

`F2008` **SINH**((1.000000, 0.000000)) has the value (1.175201, 0.000000), approximately. `F2008`

Specific Name	Argument Type	Result Type	Pass As Arg?
SINH **1**	default real	default real	yes
DSINH **2**	double precision real	double precision real	yes
QSINH **2** **3**	REAL(16)	REAL(16)	yes
Note: **1** abs(X) must be less than or equal to 89.4159. **2** abs(X) must be less than or equal to 709.7827. **3** IBM extension			

SIZE(ARRAY, DIM, KIND)

Purpose

Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class

Inquiry function

Argument type and attributes

ARRAY
> An array of any data type TS 29113 or an assumed-rank object TS 29113 . The corresponding actual argument must not be a scalar, disassociated pointer, or allocatable array that is not allocated. The actual argument can be an assumed-size array if **DIM** is present and has a value that is less than the rank of **ARRAY**.

DIM (optional)
> An INTEGER scalar. Its value must be in the range 1 ≤ **DIM** ≤ **RANK(ARRAY)**. TS 29113 It must not be present if **ARRAY** is an assumed-rank object that is associated with a scalar. TS 29113

Fortran 2003

KIND (optional)
> An INTENT(IN) INTEGER scalar. Its value must be specified by a constant expression. F2003

End of Fortran 2003

Result type and attributes

- It is of type scalar integer.

Fortran 2003

- If **KIND** is present, the **KIND** type parameter is that specified by the value of **KIND**; otherwise, the **KIND** type parameter is that of default integer type.

End of Fortran 2003

Result value

The result equals the extent of **ARRAY** along dimension **DIM**; or, if **DIM** is not specified, it is the total number of array elements in **ARRAY**.

TS 29113

- If **ARRAY** is an assumed-rank object that is associated with a scalar, the result is 1.
- If **ARRAY** is an assumed-rank object that is associated with an assumed-size array, and
  - If **DIM** is present and equal to the rank of **ARRAY**, the result is -1.
  - If **DIM** is not present, the result is a negative value that is equal to **PRODUCT([(SIZE(ARRAY, I, KIND), I=1, RANK(ARRAY))])**.

TS 29113

Examples

```
! A is the array   | 1 -4  7 -10 |
!                   | 2  5 -8  11 |
!                   | 3  6  9 -12 |
```

```
      RES = SIZE( A )
! The result is 12 because there are 12 elements in A.

      RES = SIZE( A, DIM = 1)
! The result is 3 because there are 3 rows in A.

      RES = SIZE( A, DIM = 2)
! The result is 4 because there are 4 columns in A.
```

SIZEOF(A) (IBM extension)

Purpose

Returns the size of an argument in bytes.

Class

Inquiry function

Argument type and attributes

A   An INTENT(IN) argument. The corresponding actual argument cannot be an assumed-size array.

Result type and attributes

Default integer scalar.

Result value

The size of the argument in bytes.

The size of a derived object or record structure containing an allocatable or Fortran 90 pointer component includes only the size of the deallocated object or unassociated pointer component, even if the component is allocated or associated.

Restriction: SIZEOF must not be passed as an argument to a subprogram.

Examples

The following example assumes that **–qintsize**=4.
```
INTEGER ARRAY(10)
INTEGER*8, PARAMETER :: p = 8
STRUCTURE /STR/
  INTEGER I
  COMPLEX C
END STRUCTURE
RECORD /STR/ R
CHARACTER*10 C
TYPE DTYPE
  INTEGER ARRAY(10)
END TYPE
TYPE (DTYPE) DOBJ
PRINT *, SIZEOF(ARRAY), SIZEOF (ARRAY(3)), SIZEOF(P) ! Array, array
                                                     ! element ref,
                                                     ! named constant

PRINT *, SIZEOF (R), SIZEOF(R.C)                     ! record structure
                                                     ! entity, record
                                                     ! structure
```

```
                                                  ! component

      PRINT *, SIZEOF (C(2:5)), SIZEOF(C)          ! character
                                                  ! substring,
                                                  ! character
                                                  ! variable

      PRINT *, SIZEOF (DOBJ), SIZEOF(DOBJ%ARRAY)   ! derived type
                                                  ! object, structure
                                                  ! component
```

See the following sample output generated by the program above:

```
40   4   8
16   8
 4  10
40  40
```

Related information

See the *XL Fortran Compiler Reference* for details about the **-qintsize** compiler option.

SPACING(X)
Purpose

Returns the absolute spacing of the model numbers near the argument value.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

If X is not 0, the result is:

$2.0^{\text{EXPONENT(X) - DIGITS(X)}}$

If X is 0, the result is the same as that of TINY(X).

Examples

IBM extension

SPACING $(3.0) = 2.0^{2\ -\ 24} = 2.0^{(-22)}$ See "Real data model" on page 571.

End of IBM extension

SPREAD(SOURCE, DIM, NCOPIES)

Purpose

Replicates an array in an additional dimension by making copies of existing elements along that dimension.

Class

Transformational function

Argument type and attributes

SOURCE
> An INTENT(IN) argument. The corresponding actual argument can be an array or scalar of any data type. The rank of **SOURCE** has a maximum value of 19.

DIM An INTEGER scalar. Its value must be in the range
1 ≤ DIM ≤ rank(SOURCE)+1. Unlike most other array intrinsic functions, **SPREAD** requires the DIM argument.

NCOPIES
> An INTEGER scalar. It becomes the extent of the extra dimension added to the result.

Result type and attributes

The result is an array of rank rank(SOURCE)+1 and with the same type and type parameters as source.

Result value

If SOURCE is a scalar, the result is a one-dimensional array with NCOPIES elements, each with value SOURCE.

If SOURCE is an array, the result is an array of rank rank(SOURCE) + 1. Along dimension DIM, each array element of the result is equal to the corresponding array element in SOURCE.

If NCOPIES is less than or equal to zero, the result is a zero-sized array.

Examples

```
! A is the array (/ -4.7, 6.1, 0.3 /)

      RES = SPREAD( A, DIM = 1, NCOPIES = 3 )
! The result is    | -4.7 6.1 0.3 |
!                  | -4.7 6.1 0.3 |
!                  | -4.7 6.1 0.3 |
! DIM=1 extends each column.  Each element in RES(:,1)
! becomes a copy of A(1), each element in RES(:,2) becomes
! a copy of A(2), and so on.

      RES = SPREAD( A, DIM = 2, NCOPIES = 3 )
! The result is    | -4.7 -4.7 -4.7 |
!                  |  6.1  6.1  6.1 |
!                  |  0.3  0.3  0.3 |
! DIM=2 extends each row.  Each element in RES(1,:)
! becomes a copy of A(1), each element in RES(2,:)
! becomes a copy of A(2), and so on.
```

```
         RES = SPREAD( A, DIM = 2, NCOPIES = 0 )
! The result is (/ /) (a zero-sized array).
```

SQRT(X)

Purpose

Square root.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL or COMPLEX. Unless X is COMPLEX, its value must be greater than or equal to zero.

Result type and attributes

Same as X.

Result value

- It has a value equal to the square root of X.

The **-qxlf2003=signdzerointr** option controls whether you get Fortran 2003 behavior. See **qxlf2003** in the *XL Fortran Compiler Reference*

- If the result type is complex, its value is the principal value with the real part greater than or equal to zero. If the real part is zero, the imaginary part is greater than or equal to zero.

Fortran 2003

- If the result type is complex, its value is the principal value with the real part greater than or equal to zero. If the real part of the result is zero, the imaginary part has the same sign as the imaginary part of X.

End of Fortran 2003

Examples

SQRT (4.0) has the value 2.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
SQRT	default real	default real	yes
DSQRT	double precision real	double precision real	yes
QSQRT	REAL(16)	REAL(16)	yes **1**
CSQRT **2**	default complex	default complex	yes
CDSQRT **2**	double complex	double complex	yes **1**
ZSQRT **2**	COMPLEX(8)	COMPLEX(8)	yes **1**
CQSQRT **2**	COMPLEX(16)	COMPLEX(16)	yes **1**

Specific Name	Argument Type	Result Type	Pass As Arg?
Note: **1** IBM extension: the ability to pass the name as an argument. **2** Given that X is a complex number in the form a + bi, where i = $(-1)^{\frac{1}{2}}$, abs(X) + abs(a) must be less than or equal to 1.797693 * 10^{308}			

SRAND(SEED) (IBM extension)

Purpose

Provides the seed value used by the random number generator function **RAND**. This intrinsic subroutine is not recommended. Use the standards conforming **RANDOM_NUMBER(HARVEST)** intrinsic subroutine.

Class

Subroutine

Argument type and attributes

SEED

- An INTENT(IN) REAL(4) scalar when used to provide a seed value for the **RAND** function
- An INTENT(IN) INTEGER(4) scalar when used to provide a seed value for the **IRAND** service and utility function.

Examples

See the following example of a program using the **SRAND** subroutine.

```
CALL SRAND(0.5)
DO I = 1, 5
   R = RAND()
   PRINT *,R
ENDDO
END
```

See the following sample output generated by the above program:

```
0.3984375000
0.4048461914
0.1644897461
0.1281738281E-01
0.2313232422E-01
```

STORAGE_SIZE (A, KIND) (Fortran 2008)

Purpose

Returns the storage size in bits for an element of an array that has the dynamic type and type parameters of A.

Class

Inquiry function

Argument type and attributes

A   A data object of any type.

If A is polymorphic, it cannot be an undefined pointer. If A is unlimited polymorphic or has any deferred type parameters, it cannot be an allocatable variable that is not allocated, or a pointer that is disassociated or undefined.

KIND (optional)
An `INTENT(IN)` scalar `INTEGER` . The actual argument corresponding to **KIND** must be a constant expression.

Result type and attributes

- The result is a scalar of type integer.
- If **KIND** is present, the kind type parameter is that specified by **KIND**; otherwise, the kind type parameter is that of the default integer type.

Result value

The result value is the size in bits for an element of an array that has the dynamic type and type parameters of A.

Example

The following example shows that i is assigned the value of default integer size.

```
PROGRAM main

 IMPLICIT NONE

 INTEGER i, j

 i = STORAGE_SIZE(j)
 PRINT i

END
```

SUM(ARRAY, DIM, MASK) or SUM(ARRAY, MASK)
Purpose

Calculates the sum of selected elements in an array.

Class

Transformational function

Argument type and attributes

ARRAY
An `NUMERIC` array whose elements you want to sum.

DIM (optional)
An `INTEGER` scalar. Its value must be in the range
$1 \le DIM \le rank(ARRAY)$.

MASK (optional)
A logical expression. If it is an array, it must conform with ARRAY in shape. If MASK is a scalar, the scalar value applies to all elements in ARRAY.

Result value

If DIM is present, the result is an array of rank rank(ARRAY)-1, with the same data type and kind type parameter as ARRAY. If DIM is missing, or if MASK has a rank of one, the result is a scalar.

The result is calculated by one of the following methods:

Method 1:
>If only ARRAY is specified, the result equals the sum of all the array elements of ARRAY. If ARRAY is a zero-sized array, the result equals zero.

Method 2:
>If ARRAY and MASK are both specified, the result equals the sum of the array elements of ARRAY that have a corresponding array element in MASK with a value of .TRUE.. If MASK has no elements with a value of .TRUE., the result is equal to zero.

Method 3:
>If DIM is also specified, the result value equals the sum of the array elements of ARRAY along dimension DIM that have a corresponding true array element in MASK.

Because both **DIM** and **MASK** are optional, various combinations of arguments are possible. When the **-qintlog** option is specified with two arguments, the second argument refers to one of the following:

- **MASK** if it is an array of type integer, logical, byte or typeless
- **DIM** if it is a scalar of type integer, byte or typeless
- **MASK** if it is a scalar of type logical

Examples

Method 1:
```
! Sum all the elements in an array.
      RES = SUM( (/2, 3, 4 /) )
! The result is 9 because (2+3+4) = 9
```

Method 2:
```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Sum all elements that are greater than -5.
      RES = SUM( A, MASK = A .GT. -5 )
! The result is 2 because (-3 + 2 + 3) = 2
```

Method 3:
```
! B is the array | 4 2 3 |
!                | 7 8 5 |

! Sum the elements in each column.
      RES = SUM(B, DIM = 1)
! The result is | 11 10 8 | because (4 + 7) = 11
!                                   (2 + 8) = 10
!                                   (3 + 5) =  8

! Sum the elements in each row.
      RES = SUM(B, DIM = 2)
! The result is | 9 20 | because (4 + 2 + 3) =  9
!                                (7 + 8 + 5) = 20

! Sum the elements in each row, considering only
! those elements greater than two.
```

```
      RES = SUM(B, DIM = 2, MASK = B .GT. 2)
! The result is | 7 20 | because (4 + 3) =  7
!                               (7 + 8 + 5) = 20
```

SYSTEM(CMD, RESULT) (IBM extension)

Purpose

Passes a command to the operating system for execution. The current process pauses until the command is completed and control is returned from the operating system. An added, optional argument to the subroutine will allow recovery of any return code information from the operating system.

Class

Subroutine

Argument type and attributes

CMD An INTENT(IN) CHARACTER scalar. It specifies the command to execute and any command-line arguments.

RESULT (optional)
An INTENT(OUT) INTEGER(4) scalar. If the argument is not an **INTEGER(4)** variable, the compiler generates an **(S)** level error message. The format of the information returned in **RESULT** is the same as the format returned from the wait system call.

Examples

```
      INTEGER      ULIMIT
      CHARACTER(32)  CMD
      ...
! Check the system ulimit.
      CMD = 'ulimit > ./fort.99'
      CALL SYSTEM(CMD)
      READ(99, *) ULIMIT
      IF (ULIMIT .LT. 2097151) THEN
         ...
      INTEGER RC
      RC=99
      CALL SYSTEM("/bin/test 1 -EQ 2",RC)
      IF (IAND(RC,'ff'z) .EQ. 0) then
        RC = IAND( ISHFT(RC,-8), 'ff'z )
      ELSE
        RC = -1
      ENDIF
```

SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

Purpose

Returns numeric data from a real-time clock.

Class

Subroutine

Argument type and attributes

COUNT (optional)
An INTENT(OUT) INTEGER. The initial value of COUNT depends on the

current value of the processor clock in a range from 0 to COUNT_MAX. COUNT increments by one for each clock count until it reaches the value of COUNT_MAX. At the next clock count after COUNT_MAX, the value of COUNT resets to zero.

COUNT_RATE (optional)

An INTENT(OUT) INTEGER or REAL scalar. When using the default centisecond resolution, COUNT_RATE refers to the number of processor clock counts per second or to zero if there is no clock.

▶IBM◀ If you specify a microsecond resolution using **–qsclk=micro**, the value of COUNT_RATE is 1 000 000 clock counts per second. ▶IBM◀

COUNT_MAX (optional)

An INTENT(OUT) INTEGER scalar. When using the default centisecond resolution, COUNT_MAX is the maximum number of clock counts for a given processor clock.

▶IBM◀ If you specify a microsecond resolution using **-qsclk=micro** and **COUNT_MAX** is of type **INTEGER(4)**, the value of COUNT_MAX is 1 799 999 999 clock counts, or about 30 minutes.

If you specify a microsecond resolution using **-qsclk=micro** and **COUNT_MAX** is of type **INTEGER(8)**, the value of COUNT_MAX is 86 399 999 999 clock counts, or about 24 hours. ▶IBM◀

Examples

▶IBM◀ In the following example, the clock is a 24-hour clock. After the call to SYSTEM_CLOCK, the COUNT contains the day time expressed in clock ticks per second. The number of ticks per second is available in the COUNT_RATE. The COUNT_RATE value is implementation dependent.

```
INTEGER, DIMENSION(8) :: IV
TIME_SYNC: DO
CALL DATE_AND_TIME(VALUES=IV)
IHR  = IV(5)
IMIN = IV(6)
ISEC = IV(7)
CALL SYSTEM_CLOCK(COUNT=IC, COUNT_RATE=IR, COUNT_MAX=IM)
CALL DATE_AND_TIME(VALUES=IV)

IF ((IHR == IV(5)) .AND. (IMIN == IV(6)) .AND. &
  (ISEC == IV(7))) EXIT TIME_SYNC

END DO TIME_SYNC

IDAY_SEC = 3600*IHR + IMIN*60 + ISEC
IDAY_TICKS = IDAY_SEC * IR

IF (IDAY_TICKS /= IC) THEN
  STOP 'clock error'
ENDIF
END
```

◀IBM▶

TAN(X)

Purpose

Tangent function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL `F2008` or type complex. `F2008`

Result type and attributes

Same as X.

Result value

The result value approximates tan(X).
- If X is of type real, it is considered a value in radians.
- `F2008` If X is of type complex, its real part is considered a value in radians. `F2008`

Examples

TAN(1.0) has the value 1.5574077, approximately.

`F2008` **TAN**((1.000000, 0.000000)) has the value (1.557408, 0.000000), approximately. `F2008`

Specific Name	Argument Type	Result Type	Pass As Arg?
TAN	default real	default real	yes
DTAN	double precision real	double precision real	yes
QTAN	REAL(16)	REAL(16)	yes **1**
Note: **1** IBM extension: the ability to pass the name as an argument.			

TAND(X) (IBM extension)
Purpose

Tangent function. Argument in degrees.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

Same as X.

Result value

The result approximates tan(X), where X has a value in degrees.

Examples

TAND (45.0) has the value 1.0.

Specific Name	Argument Type	Result Type	Pass As Arg?
TAND	default real	default real	yes
DTAND	double precision real	double precision real	yes
QTAND	REAL(16)	REAL(16)	yes

TANH(X)
Purpose

Hyperbolic tangent function.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) REAL `F2008` or type complex. `F2008`

Result type and attributes

Same as X.

Result value

The result value approximates tanh(X).

`F2008` If X is of type complex, its imaginary part is considered a value in radians. `F2008`

Examples

TANH(1.0) has the value 0.76159416, approximately.

`F2008` **TANH**((1.000000, 0.000000)) has the value (0.761594, 0.000000), approximately. `F2008`

Specific Name	Argument Type	Result Type	Pass As Arg?
TANH	default real	default real	yes
DTANH	double precision real	double precision real	yes
QTANH	REAL(16)	REAL(16)	yes [1]
Note: [1] IBM extension: the ability to pass the name as an argument.			

TINY(X)
Purpose

Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function

Argument type and attributes

X An INTENT(IN) real. The corresponding actual argument can be scalar or array.

Result type and attributes

Scalar with the same type and kind type parameter as X.

Result value

IBM The result is:
$2.0^{(MINEXPONENT(X)-1)}$ for real X

IBM

Examples

IBM **TINY** $(X) = float(2)^{(-126)} = 1.17549351e-38$. See "Real data model" on page 571. **IBM**

TRAILZ(I) (Fortran 2008)
Purpose

Returns the number of trailing zero bits in the binary representation of an integer.

Class

Elemental function

Argument type and attributes

I An INTENT(IN) INTEGER

Result type and attributes

Same as I.

Result value

The result is the count of zero bits to the right of the rightmost one bit for I. If I has the value zero, the result is **BIT_SIZE(I)**.

Examples

```
I = TRAILZ(0_4)  ! I=32
J = TRAILZ(4_4)  ! J=2
K = TRAILZ(-1)   ! K=0
M = TRAILZ(0_8)  ! M=64
N = TRAILZ(1_8)  ! N=0
```

Related information
- "BIT_SIZE(I)" on page 591
- "LEADZ(I) (Fortran 2008)" on page 643

TRANSFER(SOURCE, MOLD, SIZE)
Purpose

Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

It performs a low-level conversion between types without any sign extension, rounding, blank padding, or other alteration that may occur using other methods of conversion.

Class

Transformational function

Argument type and attributes

SOURCE
> The data entity whose bitwise value you want to transfer to a different type. It may be of any type, and may be a scalar or an array.

MOLD
> A data entity that has the type characteristics you want for the result. If the actual argument corresponding to **MOLD** is a variable, the value does not need to be defined. The actual argument can be of any type, and can be a scalar or an array. Its value is not used, only its type characteristics are used.

SIZE (optional)
> The number of elements for the output result. It must be a scalar integer. The corresponding actual argument must not be an optional dummy argument.

Result type and attributes

The same type and type parameters as MOLD.

If MOLD is a scalar and SIZE is absent, the result is a scalar.

If MOLD is array valued and SIZE is absent, the result is array valued and of rank one, with the smallest size that is physically large enough to hold SOURCE.

If SIZE is present, the result is array valued of rank one and size SIZE.

Result value

The physical representation of the result is the same as SOURCE, truncated if the result is smaller or with an undefined trailing portion if the result is larger.

Because the physical representation is unchanged, it is possible to undo the results of TRANSFER as long as the result is not truncated:

```
REAL(4) X /3.141/
DOUBLE PRECISION I, J(6) /1,2,3,4,5,6/
```

```
! Because x is transferred to a larger representation
! and then back, its value is unchanged.
      X = TRANSFER( TRANSFER( X, I ), X )

! j is transferred into a real(4) array large enough to
! hold all its elements, then back into an array of
! its original size, so its value is unchanged too.
      J = TRANSFER( TRANSFER( J, X ), J, SIZE=SIZE(J) )
```

Examples

IBM **TRANSFER** (1082130432, 0.0) is 4.0. **IBM**

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/)) is a complex rank-one array of length two whose first element has the value (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is undefined.

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/), 1) has the value (/(1.1,2.2)/).

TRANSPOSE(MATRIX)

Purpose

Transposes a two-dimensional array, turning each column into a row and each row into a column.

Class

Transformational function

Argument type and attributes

MATRIX
> An array of any data type, with a rank of two.

Result value

The result is a two-dimensional array of the same data type and type parameters as MATRIX.

The shape of the result is (n,m) where the shape of MATRIX is (m,n). For example, if the shape of MATRIX is (2,3), the shape of the result is (3,2).

Each element (i,j) in the result has the value MATRIX (j,i) for i in the range 1-n and j in the range 1-m.

Result type and attributes

A two-dimensional array of the same data type and type parameters as MATRIX.

Examples

```
! A is the array  | 0 -5  8 -7 |
!                 | 2  4 -1  1 |
!                 | 7  5  6 -6 |
! Transpose the columns and rows of A.
      RES = TRANSPOSE( A )
```

```
! The result is   │  0  2  7 │
!                 │ -5  4  5 │
!                 │  8 -1  6 │
!                 │ -7  1 -6 │
```

TRIM(STRING)
Purpose

Returns the argument with trailing blank characters removed.

Class

Transformational function

Argument type and attributes

STRING
        An INTENT(IN) CHARACTER scalar

Result type and attributes

Character with the same kind type parameter value as STRING and with a length that is the length of STRING less the number of trailing blanks in STRING.

Result value

- The value of the result is the same as STRING, except trailing blanks are removed.
- If STRING contains no nonblank characters, the result has zero length.

Examples

TRIM (' A B   ') has the value ' A B'.

UBOUND(ARRAY, DIM, KIND)
Purpose

Returns the upper bound of each dimension in an array, or the upper bound of a specified dimension.

Class

Inquiry function

Argument type and attributes

ARRAY
        The array whose upper bounds you want to determine. The bounds of the array must be defined; that is, the corresponding actual argument cannot be a disassociated pointer or an allocatable array that is not allocated, and if its size is assumed, you can only examine one dimension. If **ARRAY** is an assumed-size array, **DIM** shall be present with a value less than the rank of **ARRAY**.

DIM (optional)
        An INTEGER scalar. Its value must be in the range $1 \leq$ **DIM** $\leq$ **RANK(ARRAY)**. The corresponding actual argument must not be an

optional dummy argument. `TS 29113` The **DIM** argument cannot be present if **ARRAY** is an assumed-rank object that is associated with a scalar. `TS 29113`

`F2003` **KIND (optional)**
An `INTEGER` scalar. The actual argument corresponding to **KIND** must be a constant expression. `F2003`

Result type and attributes
- The result is of type integer
- `F2003` If **KIND** is present, the kind type parameter is that specified by the value of **KIND**; otherwise, the kind type parameter is that of the default integer type. `F2003`
- If **DIM** is present, the result is a scalar. If it is not present, the result is a one-dimensional array with one element for each dimension in **ARRAY**.
- `TS 29113` If **ARRAY** is an assumed-rank object that is associated with a scalar, the result is a zero-sized array. `TS 29113`
- `TS 29113` If **ARRAY** is an assumed-rank object that is associated with an assumed-size array, the result of **UBOUND(ARRAY**, **RANK(ARRAY)**, **KIND)** equals**LBOUND(ARRAY**, **RANK(ARRAY)**, **KIND)** minus two. In this context, if **KIND** is omitted for **UBOUND(ARRAY**, **RANK(ARRAY)**, **KIND)**, it is also omitted for the corresponding **LBOUND(ARRAY**, **RANK(ARRAY)**, **KIND)**. `TS 29113`

Result value

Each element in the result corresponds to a dimension of **ARRAY**. If **ARRAY** is a whole array or array structure component, these values are equal to the upper bounds. If **ARRAY** is an array section or expression that is not a whole array or array structure component, the values represent the number of elements in each dimension, which may be different than the declared upper bounds of the original array. If a dimension is zero-sized, the corresponding element in the result is zero, regardless of the value of the upper bound.

Examples
```
! This array illustrates the way UBOUND works with
! different ranges for dimensions.
       REAL A(1:10, -4:5, 4:-5)

       RES=UBOUND( A )
! The result is (/ 10, 5, 0 /).

       RES=UBOUND( A(:,:,:) )
! The result is (/ 10, 10, 0 /) because the argument
! is an array section.

       RES=UBOUND( A(4:10,-4:1,:) )
! The result is (/ 7, 6, 0 /), because for an array section,
! it is the number of elements in the corresponding dimensions.
```

UNPACK(VECTOR, MASK, FIELD)
Purpose

Takes some or all elements from a one-dimensional array and rearranges them into another, possibly larger, array.

Class

Transformational function

Argument type and attributes

VECTOR
> A one-dimensional array of any data type. There must be at least as many elements in VECTOR as there are .TRUE. values in MASK.

MASK
> A `LOGICAL` array. It determines where the elements of VECTOR are placed when they are unpacked.

FIELD An argument that has the same shape as the mask argument, and the same data type and type parameters as VECTOR. Its elements are inserted into the result array wherever the corresponding MASK element has the value .FALSE..

Result value

The elements of the result are filled in array-element order: if the corresponding element in MASK is .TRUE., the result element is filled by the next element of VECTOR; otherwise, it is filled by the corresponding element of FIELD.

Result type and attributes

An array with the same shape as MASK and the same data type and type parameters as VECTOR.

Examples

```
! VECTOR is the array (/ 5, 6, 7, 8 /),
! MASK is │ F T T │, FIELD is │ -1 -4 -7 │
!         │ T F F │           │ -2 -5 -8 │
!         │ F F T │           │ -3 -6 -9 │

! Turn the one-dimensional vector into a two-dimensional
! array. The elements of VECTOR are placed into the .TRUE.
! positions in MASK, and the remaining elements are
! made up of negative values from FIELD.
      RES = UNPACK( VECTOR, MASK, FIELD )
! The result is │ -1  6  7 │
!               │  5 -5 -8 │
!               │ -3 -6  8 │

! Do the same transformation, but using all zeros for the
! replacement values of FIELD.
      RES = UNPACK( VECTOR, MASK, FIELD = 0 )
! The result is │ 0 6 7 │
!               │ 5 0 0 │
!               │ 0 0 8 │
```

VERIFY(STRING, SET, BACK, KIND)
Purpose

Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

Class

Elemental function

Argument type and attributes

STRING
An `INTENT(IN)` `CHARACTER`

SET An `INTENT(IN)` `CHARACTER` with the same kind type parameter as STRING.

BACK (optional)
An `INTENT(IN)` `LOGICAL`

`F2003` **KIND (optional)**
An `INTENT(IN)` `INTEGER` scalar. The actual argument corresponding to
KIND must be a constant expression. `F2003`

Result type and attributes

- `F2003` It is of type integer
- If **KIND** is present, the **KIND** type parameter is that specified by the value of
  KIND; otherwise, the **KIND** type parameter is that of default integer type.
  `F2003`

Result value

- Case (i): If BACK is absent or present with the value .FALSE. and if STRING
  contains at least one character that is not in SET, the value of the result is the
  position of the leftmost character of STRING that is not in SET.
- Case (ii): If BACK is present with the value .TRUE. and if STRING contains at
  least one character that is not in SET, the value of the result is the position of the
  rightmost character of STRING that is not in SET.
- Case (iii): The value of the result is zero if each character in STRING is in SET or
  if STRING has zero length.

Examples

- Case (i): VERIFY ('ABBA', 'A') has the value 2.
- Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.
- Case (iii): VERIFY ('ABBA', 'AB') has the value 0.

Chapter 15. Hardware-specific intrinsic procedures (IBM extension)

This section provides an alphabetical reference to the hardware-specific intrinsic functions. Many of these intrinsics provide access to hardware instructions that may not strictly conform to all IEEE floating-point semantic rules depending on their usage. You should exercise caution if strict IEEE floating-point conformance is important to your application. Unless otherwise noted, an intrinsic procedure will function on any supported hardware.

Cryptography procedures

Advanced Encryption Standard procedures

VCIPHER(STATE_ARRAY, ROUND_KEY)
Purpose

Performs one round of the AES cipher operation, as defined in Federal Information Processing Standards Publication 197 (FIPS-197), on intermediate state **STATE_ARRAY** using a given **ROUND_KEY**.

Class

Elemental function

Argument type and attributes

STATE_ARRAY
> An `INTENT(IN) UNSIGNED(1)` vector.

ROUND_KEY
> An `INTENT(IN) UNSIGNED(1)` vector.

Result type and attributes

An `UNSIGNED(1)` vector.

Result value

Returns the resulting intermediate state.

VCIPHERLAST(STATE_ARRAY, ROUND_KEY)
Purpose

Performs the final round of the AES cipher operation, as defined in Federal Information Processing Standards Publication 197 (FIPS-197), on intermediate state **STATE_ARRAY** using a given **ROUND_KEY**.

Class

Elemental function

Argument type and attributes

STATE_ARRAY
    An `INTENT(IN) UNSIGNED(1)` vector.

ROUND_KEY
    An `INTENT(IN) UNSIGNED(1)` vector.

Result type and attributes

An `UNSIGNED(1)` vector.

Result value

Returns the resulting final state.

VNCIPHER(STATE_ARRAY, ROUND_KEY)
Purpose

Performs one round of the AES inverse cipher operation, as defined in Federal
Information Processing Standards Publication 197 (FIPS-197), on intermediate state
STATE_ARRAY using a given **ROUND_KEY**.

Class

Elemental function

Argument type and attributes

STATE_ARRAY
    An `INTENT(IN) UNSIGNED(1)` vector.

ROUND_KEY
    An `INTENT(IN) UNSIGNED(1)` vector.

Result type and attributes

An `UNSIGNED(1)` vector.

Result value

Returns the resulting intermediate state.

VNCIPHERLAST (STATE_ARRAY, ROUND_KEY)
Purpose

Performs the final round of the AES inverse cipher operation, as defined in Federal
Information Processing Standards Publication 197 (FIPS-197), on intermediate state
STATE_ARRAY using a given **ROUND_KEY**.

Class

Elemental function

Argument type and attributes

STATE_ARRAY
    An `INTENT(IN) UNSIGNED(1)` vector.

ROUND_KEY
> An INTENT(IN) UNSIGNED(1) vector.

Result type and attributes

An UNSIGNED(1) vector.

Result value

Returns the resulting final state.

VSBOX(STATE_ARRAY)
Purpose

Performs the SubBytes operation, as defined in Federal Information Processing Standards Publication 197 (FIPS-197), on a given **STATE_ARRAY**.

Class

Elemental function

Argument type and attributes

STATE_ARRAY
> An INTENT(IN) UNSIGNED(1) vector.

Result type and attributes

An UNSIGNED(1) vector.

Result value

Returns the result of the operation.

Secure Hash Algorithm procedures

VSHASIGMAD(X, TYPE, MASK)
Purpose

Provides support for Federal Information Processing Standards Publication FIPS-180-3, which is a specification for Secure Hash Standard.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) UNSIGNED(8) vector.

TYPE   An INTENT(IN) integer. The corresponding actual argument must be a constant expression with a value of 0 or 1. The *TYPE* selects the function type, which can be either lowercase sigma or uppercase sigma.

MASK
> An INTENT(IN) integer. The corresponding actual argument must be a constant expression with a value in the range 0 to 15 inclusive. The *MASK* selects the function subtype, which can be either sigma-0 or sigma-1.

Result type and attributes

An `UNSIGNED(8)` vector.

Result value

For each element i (i=0,1) of X, element i of the returned value is the following result SHA-512 function:

- The result SHA-512 function is `sigma0(X(i))`, if *TYPE* is 0 and bit 2*i of *MASK* is 0.
- The result SHA-512 function is `sigma1(X(i))`, if *TYPE* is 0 and bit 2*i of *MASK* is 1.
- The result SHA-512 function is `Sigma0(X(i))`, if *TYPE* is nonzero and bit 2*i of *MASK* is 0.
- The result SHA-512 function is `Sigma1(X(i))`, if *TYPE* is nonzero and bit 2*i of *MASK* is 1.

VSHASIGMAW(X, TYPE, MASK)
Purpose

Provides support for Federal Information Processing Standards Publication FIPS-180-3, which is a specification for Secure Hash Standard.

Class

Elemental function

Argument type and attributes

X An `INTENT(IN)` `UNSIGNED(4)` vector.

TYPE An `INTENT(IN)` integer. The corresponding actual argument must be a constant expression with a value of 0 or 1. The *TYPE* selects the function type, which can be either lowercase sigma or uppercase sigma.

MASK
An `INTENT(IN)` integer. The corresponding actual argument must be a constant expression with a value in the range 0 to 15, inclusive. The *MASK* selects the function subtype, which can be either sigma-0 or sigma-1.

Result type and attributes

An `UNSIGNED(4)` vector.

Result value

For each element i (i=0,1,2,3) of X, element i of the returned value is the following result SHA-256 function:

- The result SHA-256 function is `sigma0(x(i))`, if *TYPE* is 0 and bit i of *MASK* is 0.
- The result SHA-256 function is `sigma1(x(i))`, if *TYPE* is 0 and bit i of *MASK* is 1.
- The result SHA-256 function is `Sigma0(x(i))`, if *TYPE* is nonzero and bit i of *MASK* is 0.
- The result SHA-256 function is `Sigma1(x(i))`, if *TYPE* is nonzero and bit i of *MASK* is 1.

Miscellaneous procedures

VPERMXOR(A, B, MASK)
Purpose

Applies a permute and exclusive-OR operation on two byte vectors.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN)` `UNSIGNED(1)` vector.

B   An `INTENT(IN)` `UNSIGNED(1)` vector.

MASK
    An `INTENT(IN)` `UNSIGNED(1)` vector.

Result type and attributes

An `UNSIGNED(1)` vector.

Result value

For each i (0 <= i < 16), let `indexA` be bits 0 - 3 and `indexB` be bits 4 - 7 of byte element i of *MASK*.

Byte element i of the result is set to the exclusive-OR of byte elements `indexA` of *A* and `indexB` of *B*.

Related information:

 -qaltivec

 Vector element order toggling

VPMSUMB(A, B)
Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Class

Elemental function

Argument type and attributes

A   An `INTENT(IN)` `UNSIGNED(1)` vector.

B   An `INTENT(IN)` `UNSIGNED(1)` vector.

Result type and attributes

An `UNSIGNED(1)` vector.

Result value

For each i (0 <= i < 16), let prod(i) be the result of polynomial multiplication of byte elements i of *A* and *B*.

For each i (0 <= i < 8), each halfword element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 15 are set to prod(2*i) (xor) prod(2*i+1).

VPMSUMD(A, B)
Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) UNSIGNED(8) vector.

B   An INTENT(IN) UNSIGNED(8) vector.

Result type and attributes

An UNSIGNED(8) vector.

Result value

For each i (0 <= i < 2), let prod(i) be the result of polynomial multiplication of doubleword elements i of *A* and *B*.

Bit 0 of the result is set to 0.

Bits 1 - 127 of the result are set to prod(0) (xor) prod(1).

VPMSUMH(A, B)
Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) UNSIGNED(2) vector.

B   An INTENT(IN) UNSIGNED(2) vector.

Result type and attributes

An UNSIGNED(2) vector.

Result value

For each i (0 <= i < 8), let prod(i) be the result of polynomial multiplication of halfword elements i of *A* and *B*.

For each i (0 <= i < 4), each word element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 31 are set to prod(2*i) (xor) prod(2*i+1).

VPMSUMW(A, B)
Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Class

Elemental function

Argument type and attributes

A   An INTENT(IN) UNSIGNED(4) vector.

B   An INTENT(IN) UNSIGNED(4) vector.

Result type and attributes

An UNSIGNED(4) vector.

Result value

For each i (0 <= i < 4), let prod(i) be the result of polynomial multiplication of word elements i of *A* and *B*.

For each i (0 <= i < 2), each doubleword element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 63 are set to prod(2*i) (xor) prod(2*i+1).

BPERMD(MASK, SOURCE)

Purpose

Bit Permute Doubleword

Returns the result of a bit permutation operation.

Class

Function

Argument type and attributes

MASK
        An INTENT(IN) INTEGER(8)

SOURCE
        An INTENT(IN) INTEGER(8)

Result type and attributes

The result is an INTEGER(8).

Result value

Eight bits are returned, each corresponding to a bit within *SOURCE*, and were selected by a byte of *MASK*. If byte *i* of *MASK* is less than 64, the permuted bit *i* is set to the bit of source specified by byte *i* of *MASK*; otherwise the permuted bit *i* is set to 0. The permuted bits are placed in the least-significant byte of the result value and the remaining bits are filled with 0s.

Examples

If byte 2 of *MASK* has a value of 53, which is less than 64, the 53rd bit within *SOURCE* is copied into the second bit of *RES*. However; if byte 4 of *MASK* has a value of 68, which is greater than 64, 0 is copied into the fourth bit of *RES*.

```
INTEGER(8) :: MASK, SOURCE, RES
RES = BPERMD(MASK, SOURCE)
END
```

CMPB(X,Y)

Purpose

Compare Bytes

Compares each of the eight bytes of *X* with the corresponding byte of *Y*.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER(8)

Y   An INTENT(IN) INTEGER(8)

Result type and attributes

The result is an INTEGER(8).

Result value

If byte *i* of *X* and byte i of *Y* are equal, 0xFF is placed in the corresponding byte of the result; otherwise, 0x00 is placed in the corresponding byte of the result.

Examples

.
```
INTEGER(8) :: X, Y, RES
X = 11259375 ! X = 0xABCDEF
Y = 11268045 ! Y = 0xABEFCD
RES = CMPB ( X, Y ) ! RES = 0xFF0000
END
```

DIVDE(X,Y)

Purpose

Divide Doubleword Extended

Returns the result of a doubleword extended division.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER(8)

Y   An INTENT(IN) INTEGER(8)

Result type and attributes

The result is an INTEGER(8).

Result value

The result has a value equal to X/Y.

Note: If the result of the division is larger than 32 bits or if the divisor is 0, the return value of the function is undefined.

DIVWE(X,Y)

Purpose

Divide Word Extended

The result of a word extended division.

Class

Elemental function

Argument type and attributes

X   An INTENT(IN) INTEGER(4)

Y   An INTENT(IN) INTEGER(4)

Result type and attributes

The result is an INTEGER(4).

Result value

The result has a value equal to X/Y.

Note: If the divisor is 0, the return value of the function is undefined.

FCFI(I)

Purpose

Floating-point Conversion from Integer

Converts an integer value in a floating-point variable into a floating-point value.

Note: FCFI is another name for FCFID. For details, see "FCFID(I)."

Class

Function

Argument type and attributes

I   An INTENT(IN) REAL(8)

Result type and attributes

The result is a REAL(8).

Result value

The double-precision floating-point value of *I*.

Examples

```
...
REAL*8 :: R8, RES
INTEGER*8 :: I8
EQUIVALENCE(R8, I8)

I8 = 89
RES = FCFI(R8) ! RES = 89.0
...
```

FCFID(I)

Purpose

Floating-point Conversion from signed Integer Double word

Converts a 64-bit signed integer stored in a double into a double-precision floating-point value.

Note: FCFID(I) is another name for "FCFI(I)."

Class

Function

Argument type and attributes

I   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as I.

FCFUD(I)

Purpose

Floating-point Conversion from Unsigned integer Double word

Converts a 64-bit unsigned integer stored in a double into a double-precision floating-point value.

Class

Function

Argument type and attributes

I An `INTENT(IN) REAL(8)`

Result type and attributes

The result is of the same type as I.

FCTID(X)

Purpose

Floating-point Convert to Integer

Converts a floating–point operand into a 64-bit, signed fixed–point integer using the current rounding mode.

Class

Function

Argument type and attributes

X   An `INTENT(IN) REAL(8)`

Result type and attributes

The result is of the same type as X.

Result value

The result is a fixed-point integer, inside a floating-point result.

Examples

```
use, intrinsic :: ieee_arithmetic
real(8) :: x, y
integer(8) :: i
equivalence (y, i)
x = 1234.5678D0
if (ieee_support_datatype(x)) then
  call ieee_set_rounding_mode(ieee_nearest)
  y = fctid(x)
  print *, i
  call ieee_set_rounding_mode(ieee_up)
  y = fctid(x)
```

```
      print *, i
      call ieee_set_rounding_mode(ieee_down)
      y = fctid(x)
      print *, i
      call ieee_set_rounding_mode(ieee_to_zero)
      y = fctid(x)
      print *, i
   endif
   end
```

The following is sample output generated by the above program:

```
1235
1235
1234
1234
```

FCTIDZ(X)

Purpose

Floating-point Convert to Integer Round to Zero

Converts a floating-point operand into a 64-bit signed fixed–point integer and rounds to zero.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is a fixed-point integer, inside a floating-point result, rounded to zero.

FCTIW(X)

Purpose

Floating-point Convert to Integer

Converts a floating–point operand into a 32–bit, signed fixed–point integer using the current rounding mode.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is a fixed-point integer, inside a floating-point result.

FCTIWZ(X)

Purpose

Floating-point Convert to Integer Round to Zero

Converts a floating-point operand into a 32–bit signed fixed–point integer and rounds to zero.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is a fixed-point integer, inside a floating-point result, rounded to zero.

FCTUDZ(X)

Purpose

Floating-point Conversion to Unsigned integer Double word with rounding towards Zero

Converts a floating-point value to unsigned integer double word and rounds to zero.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is an unsigned fixed-point integer, inside a floating-point result, rounded to zero.

FCTUWZ(X)

Purpose

Floating-point conversion to unsigned integer word with rounding to zero.

Converts a floating-point number into a 32-bit unsigned integer and rounds to zero. This function is intended for use with the STFI intrinsic subroutine.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of type REAL(8).

Result value

The result is a REAL(8) value. The low-order 32 bits of the result contain the result from converting X to a non-negative INTEGER(4) that is rounded to zero. The high-order 32 bits of the result contain an undefined value.

Example

The following example demonstrates the usage of this function.

```
IMPLICIT NONE
REAL(8) x, result
INTEGER(4) y
x = -1.5D0
result = FCTUWZ(x)
CALL SFTI(y, result)
PRINT *, y        ! prints 0

x= 1.5D0
result = FCTUWZ(x)
CALL SFTI(y, result)
PRINT *, y        ! prints 1
END
```

FMADD(A, X, Y)

Purpose

Floating-point Multiply and Add

Returns the result of a floating-point multiply-add.

Class

Function

Argument type and attributes

A   An `INTENT(IN)` `REAL(4)` or `REAL(8)`.

X   An INTENT(IN) argument of the same type and kind type parameter as *A*.

Y   An INTENT(IN) argument of the same type and kind type parameter as *A*.

Result type and attributes

The result is of the same type as *A*, *X*, and *Y*.

Result value

The result has a value equal to $A*X + Y$.

Examples

```
REAL(4) :: A, B, C, RES1
REAL(8) :: D, E, F, RES2

RES1 = FMADD(A, B, C)
RES2 = FMADD(D, E, F)
END
```

FMSUB(A, X, Y)

Purpose

Floating-point Multiply and Subtract

Returns the result of a floating-point multiply–subtract.

Class

Function

Argument type and attributes

A   An `INTENT(IN)` `REAL(4)` or `REAL(8)`.

X   An INTENT(IN) argument of the same type and kind type parameter as *A*.

Y   An INTENT(IN) argument of the same type and kind type parameter as *A*.

Result type and attributes

The result is of the same type as *A*, *X*, and *Y*.

Result value

The result has a value equal to $A*X - Y$.

FNABS(X)

Purpose

Returns the negative floating-point value $-|X|$.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL

Result type and attributes

The result is of the same type as X.

Result value

The result is a negative floating-point value of X, $-|X|$.

Examples

The absolute contents of variables A and D are negated.

```
REAL(4) :: A, RES1
REAL(8) :: D, RES2

RES1 = FNABS(A)
RES2 = FNABS(D)
```

FNMADD(A, X, Y)

Purpose

Floating-point Negative Multiply and Add

Returns the result of a floating-point negative multiply-add.

Class

Function

Argument type and attributes

A   An INTENT(IN) REAL(4) or REAL(8).

X   An INTENT(IN) argument of the same type and kind type parameter as A.

Y   An INTENT(IN) argument of the same type and kind type parameter as A.

Result type and attributes

The result is of the same type as X.

Result value

The result has a value equal to $-(A^*X + Y)$.

FNMSUB(A, X, Y)

Purpose

Floating-point Negative Multiply and Subtract

Returns the result of a floating-point negative multiply–subtract.

Class

Function

Argument type and attributes

A An INTENT(IN) REAL(8)

X An INTENT(IN) argument of the same type and kind type parameter as *A*.

Y An INTENT(IN) argument of the same type and kind type parameter as *A*.

Result type and attributes

The result is of the same type as *A*, *X*, and *Y*.

Result value

The result has a value equal to –(*A**X – *Y*).

Examples

The result of **FNMSUB** is of type **REAL(4)**. It is converted to **REAL(8)** and then assigned to *RES*.

```
REAL(4) :: A, B, C
REAL(8) :: RES

RES = FNMSUB(A, B, C)
END
```

FRE(X)

Purpose

Floating-point Reciprocal Estimate

Returns an estimate of a floating-point reciprocal operation.

Class

Function

Argument type and attributes

X An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is a double precision estimate of $1/X$.

FRES(X)

Purpose

Floating-point Reciprocal Estimate Single

Returns an estimate of a floating-point reciprocal operation

Valid on any PowerPC with extended graphics opcodes. See **Tuning for your target architecture** in the *XL Fortran Optimization and Programming Guide*.

Class

Function

Argument type and attributes

X   An `INTENT(IN) REAL(4)`

Result type and attributes

The result is of the same type as `X`.

Result value

The result is a single precision estimate of $1/X$.

FRIC(A)

Purpose

Floating-point Rounding to Integer with Current rounding mode

Rounds a double-precision floating-point value to integer with the current rounding mode.

Class

Function

Argument type and attributes

A An `INTENT(IN) REAL(8)`

Result type and attributes

The result is of the same type as `A`.

FRIM(A)

Purpose

Floating-point Round to Integer Minus

Class

Function

Argument type and attributes

A   An INTENT(IN) REAL(4) or REAL(8)

Result type and attributes

The result is of the same type as A.

Result value

The result has a value equal to the greatest integer less than or equal to *A*.

FRIN(A)

Purpose

Floating-point Round to Integer Nearest

Class

Function

Argument type and attributes

A   An INTENT(IN) REAL(4) or REAL(8)

Result type and attributes

The result is of the same type as A.

Result value

If A > 0, **FRIN(A)** has the value **FRIM(A + 0.5)**.

If A <= 0, **FRIN(A)** has the value **FRIM(A - 0.5)**.

FRIP(A)

Purpose

Floating-point Round to Integer Plus

Class

Function

Argument type and attributes

A   An INTENT(IN) REAL(4) or REAL(8)

Result type and attributes

The result is of the same type as A.

Result value

The result has a value equal to the least integer greater than or equal to *A*.

FRIZ(A)

Purpose

Floating-point Round to Integer Zero

Class

Function

Argument type and attributes

A   An INTENT(IN) REAL(4) or REAL(8)

Result type and attributes

The result is of the same type as A.

Result value

If A > 0, **FRIZ(A)** has the value **FRIM(A)**.

If A <= 0, **FRIZ(A)** has the value **FRIP(A)**.

FRSQRTE(X)

Purpose

Floating-point Square Root Reciprocal Estimate

Returns the result of a reciprocal square root operation

Valid on any PowerPC with extended graphics opcodes. See **Tuning for your target architecture** in the *XL Fortran Optimization and Programming Guide*.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(8)

Result type and attributes

The result is of the same type as X.

Result value

The result is a double precision estimate of the reciprocal of the square root of *X*.

FRSQRTES(X)

Purpose

Floating-point Square Root Reciprocal Estimate Single

Returns the result of a reciprocal square root operation.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(4)

Result type and attributes

The result is of the same type as X.

Result value

The result is a single precision estimate of the reciprocal of the square root of *X*.

FSEL(X,Y,Z)

Purpose

Floating-point Selection

Returns the result of a floating-point selection operation. This result is determined by comparing the value of X with zero.

Valid on any PowerPC with extended graphics opcodes. See **Tuning for your target architecture** in the *XL Fortran Optimization and Programming Guide*.

Class

Function

Argument type and attributes

X   An INTENT(IN) REAL(4) or REAL(8)

Y   An INTENT(IN) argument of the same type and kind type parameter as **X**.

Z   An INTENT(IN) argument of the same type and kind type parameter as **X**.

Result type and attributes

The result is of the same type as *X*, *Y* and *Z*.

Result value
- If the value of *X* is greater than or equal to zero, then the value of *Y* is returned.
- If the value of *X* is smaller than zero or is a NaN, then the value of *Z* is returned.

A zero value is considered unsigned. That is, both +0 and -0 are equal to zero.

LOAD2R(X)

Purpose

Load 2-byte integer with byte reversal.

Class

Elemental function.

Argument type and attributes

X   An INTENT(IN) INTEGER(2).

Result type and attributes

The result is of the same type and type parameters as X.

Result value

The value of X in reverse byte order.

Examples

```
INTEGER(2) x, y
x = 1
y = LOAD2R(x)
PRINT 10, x    !prints 0001
PRINT 10, y    !prints 0100

10 FORMAT (z4.4)
END
```

Related information

LOAD4R(X)

LOAD8R(X)

REVERSE_BYTE_ORDER(X)

VEC_REVB(ARG1)

VEC_REVE(ARG1)

LOAD4R(X)

Purpose

Load 4-byte integer with byte reversal.

Class

Elemental function.

Argument type and attributes

X   An `INTENT(IN) INTEGER(4)`.

Result type and attributes

The result is of the same type and type parameters as `X`.

Result value

The value of `X` in reverse byte order.

Examples

```
INTEGER(4) x, y
x = 1
y = LOAD4R(x)
PRINT 10, x     !prints 00000001
PRINT 10, y     !prints 01000000

10 FORMAT (z8.8)
END
```

Related information

LOAD2R(X)

LOAD8R(X)

REVERSE_BYTE_ORDER(X)

VEC_REVB(ARG1)

VEC_REVE(ARG1)

LOAD8R(X)

Purpose

Load 8-byte integer with byte reversal.

Class

Elemental function.

Argument type and attributes

X   An `INTENT(IN) INTEGER(8)`.

Result type and attributes

The result is of the same type and type parameters as `X`.

Result value

The value of `X` in reverse byte order.

Examples

```
INTEGER(8) x, y
x = 1
y = LOAD8R(x)
PRINT 10, x    !prints 0000000000000001
PRINT 10, y    !prints 0100000000000000

10 FORMAT (z16.16)
END
```

Related information

LOAD2R(X)

LOAD4R(X)

REVERSE_BYTE_ORDER(X)

VEC_REVB(ARG1)

VEC_REVE(ARG1)

MTFSF(MASK, R)

Purpose

Move to floating-point status and control register (**FPSCR**) fields

The contents of *R* are placed into the **FPSCR** under control of the field mask specified in *MASK*.

Class

Subroutine

Argument type and attributes

MASK
> An INTENT(IN) INTEGER(4). The actual argument corresponding to **MASK** must be a constant expression with a value in the range 0 - 7.

R   An INTENT(IN) REAL(8).

MTFSFI(BF, I)

Purpose

Move to floating-point status and control register (**FPSCR**) Fields Immediate

The value of *I* is placed into **FPSCR** field specified in *BF*.

Class

Subroutine

Argument type and attributes

BF An `INTENT(IN) INTEGER(4)`. The actual argument corresponding to **BF** must be a constant expression with a value in the range 0 - 7.

I An `INTENT(IN) INTEGER(4)`. The actual argument corresponding to **I** must be a constant expression with a value in the range 0 - 15.

MULHY(RA, RB)

Purpose

Returns the high-order 32 or 64-bits of the 64-bit or 128-bit products of the operands *RA* and *RB*.

Class

Function

Argument type and attributes

RA An `INTENT(IN) INTEGER(4)` or `INTEGER(8)`.

RB An `INTENT(IN)` argument of the same type and kind type parameter as **RA**.

Result type and attributes

The result is of the same type as *RA, RB*.

Result value

The high-order 32 or 64-bit product of the operands *RA* and *RB*

POPCNTB(I)

Purpose

Population count.

Counts the number of set bits of each byte in **I**.

Class

Elemental function.

Argument type and attributes

I

   An `INTENT(IN) INTEGER(4)` or `INTEGER(8)`.

Result type and attributes

The result is an `INTEGER(8)`.

Result value

The number of bits set to on in that byte, in the position of the byte.

Examples

```
INTEGER I
I = x'010300ff'
WRITE(*, '(z8.8)') POPCNTB(I)
END
```

Expected output:
```
01020008
```

Related information

Data representation models

REVERSE_BYTE_ORDER(X)

Purpose

Reverse the byte order of an integer value.

This function is the generic form of `load2r`, `load4`, and `load8r`.

Class

Elemental function.

Argument type and attributes

X   An `INTENT(IN)` `INTEGER(2)`, `INTEGER(4)`, or `INTEGER(8)`.

Result type and attributes

The result is of the same type and type parameters as X.

Result value

The value of X in reverse byte order.

Examples

```
INTEGER(2) y2
INTEGER(4) y4
INTEGER(8) y8
PRINT '(z4.4)', REVERSE_BYTE_ORDER(1_2)
PRINT '(z8.8)', REVERSE_BYTE_ORDER(1_4)
PRINT '(z16.16)', REVERSE_BYTE_ORDER(1_8)
END
```

Expected output:
```
0100
01000000
0100000000000000
```

Related information

LOAD2R(X)

LOAD4R(X)

LOAD8R(X)

VEC_REVB(ARG1)

VEC_REVE(ARG1)

ROTATELI(RS, IS, SHIFT, MASK)

Purpose

Rotate Left Immediate then MASK Insert

Rotates the value of *RS* left by the number of bits specified in *SHIFT*. The function then inserts *RS* into *IS* under bit mask, *MASK*.

Class

Function

Argument type and attributes

RS   An INTENT(IN) INTEGER.

IS   An INTENT(IN) INTEGER.

SHIFT
> An INTENT(IN) INTEGER. For 4-byte **RS** values, the **SHIFT** value is truncated to the last five bits. For 8-byte **RS** values, the **SHIFT** value is truncated to the last six bits.

MASK
> An INTENT(IN) INTEGER.

Result type and attributes

The result is of the same type as *RS*.

Result value

Rotates *RS* left the number of bits specified by *SHIFT*, and inserts the result into *IS* under the bit mask, *MASK*.

ROTATELM(RS, SHIFT, MASK)

Purpose

Rotate Left AND with Mask

Rotates the value of *RS* left by the number of bits specified in *SHIFT*. The rotated data is ANDed with the *MASK* and then returned as a result.

Class

Function

Argument type and attributes

RS   An INTENT(IN) INTEGER.

SHIFT

> An INTENT(IN) INTEGER. For 4-byte **RS** values, the **SHIFT** value is truncated to the last five bits. For 8-byte **RS** values, the **SHIFT** value is truncated to the last six bits.

MASK

> An INTENT(IN) INTEGER.

Result type and attributes

The result is of the same type as *RS*.

Result value

The rotated data ANDed with *MASK*.

SETFSB0(BT)

Purpose

Move 0 to floating-point status and control register (**FPSCR**) bit.

Bit *BT* of FPSCR is set to 0.

Class

Subroutine

Argument type and attributes

BT An INTENT(IN) INTEGER(4). The actual argument corresponding to **BT** must be a constant expression with a value in the range 0 - 31.

SETFSB1(BT)

Purpose

Move 1 to FPSCR bit.

Bit *BT* of FPSCR is set to 1.

Class

Subroutine

Argument type and attributes

BT An INTENT(IN) INTEGER(4). The actual argument corresponding to **BT** must be a constant expression with a value in the range 0 - 31.

SFTI(M, Y)

Purpose

Store Floating–point to Integer

The contents of the low order 32–bits of *Y* are stored without conversion into *M*.

Class

Subroutine

Argument type and attributes

M   An `INTENT(INOUT) INTEGER(4)`.

Y   An `INTENT(IN) REAL(8)`.

Examples

```
...
integer*4 :: m
real*8 :: x

x = z"00000000abcd0001"
call sfti(m, x) ! m = z"abcd0001"
..
```

SWDIV(X,Y)

Purpose

Software floating-point division.

This function returns the result of a floating-point division and can increase performance over the normal divide operator where your application performs division repeatedly within a loop.

Class

Elemental function

Argument type and attributes

X   An `INTENT(IN) REAL(4)` or `REAL(8)`.

Y   An INTENT(IN) argument of the same type and kind type parameter as *X*.

Result type and attributes

The result is of the same type as *X* and *Y*.

Result value

The result has a value equal to *X/Y*.

If programs are compiled with the **-qarch=pwr8** option and run on a POWER8 processor:
- With **-qstrict=ieeefp** or **-qstrict=exceptions** in effect, the result is bitwise identical to IEEE division.
- With **-qstrict=noieeefp** or **-qstrict=noexceptions** in effect, the result can differ slightly from the IEEE result.

SWDIV_NOCHK(X,Y)

Purpose

Software floating-point division with checking for invalid arguments.

This function returns the result of a floating-point division and can increase performance over the normal divide operator or the **SWDIV** intrinsic function where your application performs division repeatedly within a loop, and arguments are within the permitted range.

Class

Elemental function

Argument type and attributes

X An INTENT(IN) REAL(4) or REAL(8).

Y An INTENT(IN) argument of the same type and kind type parameter as X.

For correct operation, **REAL(4)** arguments must satisfy the following conditions:

- $2^{**}(-125) <= |Y| <= 2^{**}124$
- If X is not zero, then:

  $2^{**}(-102) <= |X| < \text{Inf}$

  $2^{**}(-124) <= |X/Y| <= 2^{**}126$

For correct operation, **REAL(8)** arguments must satisfy the following conditions:

- $2^{**}(-1021) <= |Y| <= 2^{**}1020$
- If X is not zero, then:

  $2^{**}(-969) <= |X| < \text{Inf}$

  $2^{**}(-1020) <= |X/Y| <= 2^{**}1022$

Result type and attributes

The result is of the same type as X and Y.

Result value

The result has a value equal to X/Y.

If programs are compiled with the **-qarch=pwr8** option and run on a POWER8 processor:

- With **-qstrict=ieeefp** or **-qstrict=exceptions** in effect, the result is bitwise identical to IEEE division.
- With **-qstrict=noieeefp** or **-qstrict=noexceptions** in effect, the result can differ slightly from the IEEE result.

TRAP(A, B, TO)

Purpose

Operand *A* is compared with operand *B*. This comparison results in five conditions which are ANDed with *TO*. If the result is not 0, the system trap handler is invoked.

Both operands *A* and *B* must be either of type **INTEGER(4)** or **INTEGER(8)**.

Class

Subroutine

Argument type and attributes

A   An INTENT(IN) INTEGER.

B   An INTENT(IN) INTEGER.

TO   An INTENT(IN) INTEGER(4). The actual argument corresponding to **TO** must be a constant expression with a value in the range 0 - 31.

Chapter 16. The TRANSACTIONAL_MEMORY intrinsic module (IBM extension)

The TRANSACTIONAL_MEMORY intrinsic module is used for parallel programming. This module provides functions that allow you to designate a block of instructions or statements to be treated atomically. Such an atomic block is called a transaction. When a thread executes a transaction, all of the memory operations within the transaction occur simultaneously from the perspective of other threads.

For some kinds of parallel programs, a transaction implementation can be more efficient than other implementation methods, such as locks. You can use these intrinsic procedures to mark the beginning and end of transactions, and to diagnose the reasons for failure.

The transactional state is entered following a successful call to `TM_BEGIN` or `TM_SIMPLE_BEGIN`, and ended by `TM_END`, `TM_ABORT`, `TM_NAMED_ABORT`, or by transaction failure.

Note: Debugging your application while it is inside a transactional state might result in unpredictable behavior.

Transaction failure occurs when any of the following conditions is met:
- Memory that is accessed in the transactional state is accessed by another thread before the transaction completes.
- The architecture-defined footprint for memory accesses within a transaction is exceeded.
- The architecture-defined nesting limit for nested transactions is exceeded.

Transactions can be nested. You can use `TM_BEGIN` or `TM_SIMPLE_BEGIN` in the transactional state. Within an outermost transaction initiated with `TM_BEGIN`, nested transactions must be initiated with `TM_SIMPLE_BEGIN`, or by `TM_BEGIN` using the same buffer of the outermost containing transaction.

A nested transaction is subsumed into the containing transaction. Therefore, a failure of the nested transaction is treated as a failure of all containing transactions, and the nested transaction completes only when all containingtransactions complete.

Type

The TRANSACTIONAL_MEMORY module provides the following derived type:

TM_BUFF_TYPE
> An opaque derived type that is used to store transaction status.

Named constant

The TRANSACTIONAL_MEMORY module provides the following named constant:

TM_SUCCESS
> An integer constant that is returned by the transaction begin and end

functions. If `TM_SUCCESS` is returned by `TM_BEGIN` or `TM_SIMPLE_BEGIN`, it indicates that the transaction was started successfully. If `TM_SUCCESS` is returned by `TM_END`, it indicates that the thread was in the transactional state when `TM_END` was called. `TM_SUCCESS` has kind type parameter value 8.

Migration consideration: The numeric value of `TM_SUCCESS` is changed since XL Fortran for Linux, V15.1.2 (Little Endian Distributions). If you use named constant `TM_SUCCESS` to check the return value of the transaction begin or end function in your code, this change does not affect the correctness of your code.

Procedures

The TRANSACTIONAL_MEMORY module provides the following procedures:
- "Transaction begin and end functions"
- "Transaction abort functions" on page 759
- "Transaction inquiry functions" on page 760

Transaction begin and end functions

TM_SIMPLE_BEGIN()
Purpose

Marks the beginning of a transaction.

Class

Function

Result type and attributes
- `INTEGER(8)`

Result value

Returns `TM_SUCCESS` from the TRANSACTIONAL_MEMORY module if the transaction is started successfully.

Note: The transaction status of transactions started using `TM_SIMPLE_BEGIN` cannot be queried using the transaction inquiry functions.

TM_BEGIN(TM_BUFF)
Purpose

Marks the beginning of a transaction.

Class

Function

Argument type and attributes

TM_BUFF
    `INTENT(OUT)` argument of type **TM_BUFF_TYPE** from the TRANSACTIONAL_MEMORY module.

Result type and attributes
- `INTEGER(8)`

Result value

Returns `TM_SUCCESS` from the TRANSACTIONAL_MEMORY module if the transaction is started successfully.

The transaction status is stored in the *TM_BUFF* argument. This status can be queried by using the transaction inquiry functions.

TM_END()
Purpose

Marks the end of a transaction.

Class

Function

Result type and attributes
- `INTEGER(8)`

Result value

Returns `TM_SUCCESS` from the TRANSACTIONAL_MEMORY module if the thread was in the transactional state when `TM_END` started.

Transaction abort functions

TM_ABORT()
Purpose

Aborts a transaction with failure code 0.

Class

Subroutine

TM_NAMED_ABORT(CODE)
Purpose

Aborts a transaction with failure code *code*.

Class

Subroutine

Argument type and attributes

code   `INTENT(IN)` integer. The value of the corresponding actual argument must be between 0 and 255, inclusive.

Transaction inquiry functions

TM_IS_USER_ABORT(TM_BUFF)
Purpose

Queries whether the transaction was aborted because of a user request.

Class

Function

Argument type and attributes

TM_BUFF
    INTENT(IN) argument of type **TM_BUFF_TYPE** from the
    TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument
was aborted by the user; otherwise, .FALSE. is returned.

TM_IS_NAMED_USER_ABORT(TM_BUFF, CODE)
Purpose

Queries whether the transaction was aborted because of a user request and gets
the failure code that is provided by the user.

Class

Function

Argument type and attributes

TM_BUFF
        INTENT(IN) argument of type **TM_BUFF_TYPE** from the
        TRANSACTIONAL_MEMORY module.

code   INTENT(OUT) integer.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument
was aborted by the user; otherwise, .FALSE. is returned. *code* is assigned the failure
code provided by the user when the transaction was aborted, or 0 if the user did
not provide a failure code.

TM_IS_ILLEGAL(TM_BUFF)

Purpose

Queries whether the transaction was aborted because of an attempt to do something illegal.

Class

Function

Argument type and attributes

TM_BUFF
> INTENT(IN) argument of type **TM_BUFF_TYPE** from the TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument was aborted because of an attempt to do something illegal, such as an instruction not permitted in transactional mode. Otherwise, .FALSE. is returned.

TM_IS_FOOTPRINT_EXCEEDED(TM_BUFF)

Purpose

Queries whether the transaction was aborted because of exceeding the maximum number of cache lines.

Class

Function

Argument type and attributes

TM_BUFF
> INTENT(IN) argument of type **TM_BUFF_TYPE** from the TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction was aborted because of exceeding the maximum number of cache lines; otherwise, .FALSE. is returned.

TM_NESTING_DEPTH(TM_BUFF)

Purpose

Gets the current nesting depth, or if not in transactional mode, the depth at which the most recent transaction was aborted.

Class

Function

Result type and attributes

Default integer

Argument type and attributes

TM_BUFF
    INTENT(IN) argument of type **TM_BUFF_TYPE** from the TRANSACTIONAL_MEMORY module.

Result value

Returns the current nesting depth of the transaction whose status is stored in the *TM_BUFF* argument, or if not in transactional mode, the depth at which the transaction was aborted. If the transaction was completed successfully, 0 is returned.

TM_IS_NESTED_TOO_DEEP(TM_BUFF)

Purpose

Queries whether the transaction was aborted because of trying to exceed the maximum nesting depth.

Class

Function

Argument type and attributes

TM_BUFF
    INTENT(IN) argument of type **TM_BUFF_TYPE** from the TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument was aborted because of trying to exceed the maximum nesting depth; otherwise, .FALSE. is returned.

TM_IS_CONFLICT(TM_BUFF)
Purpose

Queries whether the transaction was aborted because of a conflict.

Class

Function

Argument type and attributes

TM_BUFF
    INTENT(IN) argument of type **TM_BUFF_TYPE** from the
    TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Return value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument
was aborted because of a conflict; otherwise, .FALSE. is returned.

TM_IS_FAILURE_PERSISTENT(TM_BUFF)
Purpose

Queries whether the transaction was aborted because of a persistent reason.

Class

Function

Argument type and attributes

TM_BUFF
    INTENT(IN) argument of type **TM_BUFF_TYPE** from the
    TRANSACTIONAL_MEMORY module.

Result type and attributes

Default logical

Result value

Returns .TRUE. if the transaction whose status is stored in the *TM_BUFF* argument
was aborted because of a persistent reason; otherwise, .FALSE. is returned.

TM_FAILURE_ADDRESS()
Purpose

Gets the code address at which the most recent transaction was aborted.

Class

Function

Argument type and attributes

TM_BUFF
INTENT(IN) argument of type **TM_BUFF_TYPE** from the
TRANSACTIONAL_MEMORY module.

Result type and attributes

• INTEGER(8)

Result value

Returns the address at which the transaction whose status is stored in the
TM_BUFF argument was aborted.

TM_FAILURE_CODE(TM_BUFF)
Purpose

Gets the raw failure code for the transaction.

Class

Function

Argument type and attributes

TM_BUFF
INTENT(IN) argument of type **TM_BUFF_TYPE** from the
TRANSACTIONAL_MEMORY module.

Result type and attributes

INTEGER(8)

Result value

Returns the contents of the TEXASR register stored in the *TM_BUFF* argument.
You can consult the hardware specification for the information about how to
interpret the return value.

Chapter 17. Vector intrinsic procedures (IBM extension)

Individual elements of vectors can be accessed by using storage association, the **TRANSFER** intrinsic, or the Vector Multimedia Extension (VMX) and the Vector Scalar Extension (VSX) intrinsic functions. This section provides an alphabetic reference to the VMX and the VSX intrinsic functions. These intrinsics allow you to manipulate vectors.

Note:

- You must specify appropriate compiler options for your architecture when you use the intrinsic functions.
- Intrinsic functions that use or result in the **INTEGER(8)**, **UNSIGNED(8)**, or **REAL(8)** vector types require an architecture that supports the VSX instruction set extensions.

Related information:

    📄 -qaltivec

VEC_ABS(ARG1)

Purpose

Returns a vector containing the absolute values of the contents of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4), UNSIGNED(1), UNSIGNED(2), UNSIGNED(4), REAL(4), or REAL(8) vector

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the absolute value of the corresponding element of ARG1. For integer vectors, the arithmetic is modular.

VEC_ABSD(ARG1, ARG2)

Purpose

Returns a vector that contains the absolute difference of the corresponding elements of the given vectors.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1), UNSIGNED(2), or UNSIGNED(4) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is the absolute difference of the corresponding elements of ARG1 and ARG2 using the modulo arithmetic.

VEC_ABSS(ARG1)

Purpose

Returns a vector containing the saturated absolute values of the elements of a given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer vector whose elements are of kind 1, 2, or 4

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the saturated absolute value of the corresponding element of ARG1.

VEC_ADD(ARG1, ARG2)

Purpose

Returns a vector containing the sums of each set of corresponding elements of the given vectors.

Note: For INTEGER(8) vectors, this function emulates the operation.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` integer, unsigned, or real vector

ARG2
    An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is of the same type as `ARG1`.

Result value

The value of each element of the result is the sum of the corresponding elements of `ARG1` and `ARG2`. For integer vectors and unsigned vectors, the arithmetic is modular.

VEC_ADD_U128 (ARG1, ARG2)

Purpose

Adds unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

ARG2
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

Result type and attributes

Same as *ARG1*.

Result value

Let `a` and `b` be the 128-bit integers that are storage-associated with *ARG1* and *ARG2*, respectively.

The result value is `a + b`.

VEC_ADDC_U128 (ARG1, ARG2)

Purpose

Gets the carry bit of the 128-bit addition of two quadword values.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

ARG2
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

Result type and attributes

Same as *ARG1*.

Result value

Let a and b be the 128-bit integers that are storage-associated with *ARG1* and *ARG2*, respectively.

The result value is the carry out of a + b.

VEC_ADDE_U128 (ARG1, ARG2, ARG3)

Purpose

Adds unsigned quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

ARG2
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

ARG3
    An `INTENT(IN)` `VECTOR(UNSIGNED(1))`

Result type and attributes

Same as *ARG1*.

Result value

Let a, b, c be the 128-bit integers that are storage-associated with *ARG1*, *ARG2*, and *ARG3*, respectively.

The result value is `a + b + iand(c, 1)`.

VEC_ADDEC_U128 (ARG1, ARG2, ARG3)

Purpose

Gets the carry bit of the 128-bit addition of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) VECTOR(UNSIGNED(1))

ARG2
   An INTENT(IN) VECTOR(UNSIGNED(1))

ARG3
   An INTENT(IN) VECTOR(UNSIGNED(1))

Result type and attributes

Same as *ARG1*.

Result value

Let a, b, c be the 128-bit integers that are storage-associated with *ARG1*, *ARG2*, and *ARG3*, respectively.

The result value is the carry out of `a + b + iand(c, 1)`.

VEC_ADDC(ARG1, ARG2)

Purpose

Returns a vector containing the carries produced by adding each set of corresponding elements of two given vectors.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) UNSIGNED(4) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

If a carry is produced by adding the corresponding elements of ARG1 and ARG2, the corresponding element of the result is 1; otherwise, it is 0.

VEC_ADDS(ARG1, ARG2)

Purpose

Returns a vector containing the saturated sums of each set of corresponding elements of two given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, or 4

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the saturated sum of the corresponding elements of ARG1 and ARG2.

VEC_ALL_EQ(ARG1, ARG2)

Purpose

Tests whether all sets of corresponding elements of the given vectors are equal.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_GE(ARG1, ARG2)

Purpose

Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if all elements of ARG1 are greater than or equal to the corresponding elements of ARG2. Otherwise, the result is 0.

VEC_ALL_GT(ARG1, ARG2)

Purpose

Tests whether all elements of the first argument are greater than the corresponding elements of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if all elements of ARG1 are greater than the corresponding elements of ARG2. Otherwise, the result is 0.

VEC_ALL_IN(ARG1, ARG2)

Purpose

Tests whether each element of a given vector is within a given range.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) REAL(4) vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if both of the following conditions are satisfied; otherwise, the result is 0.
- All the elements of ARG2 have a value greater than or equal to 0.
- All the elements of ARG1 have a value less than or equal to the value of the corresponding element of ARG2, and greater than or equal to the negative of the value of the corresponding element of ARG2.

VEC_ALL_LE(ARG1, ARG2)

Purpose

Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) integer, unsigned, or real vector

ARG2
    An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if all elements of `ARG1` are less than or equal to the corresponding elements of `ARG2`. Otherwise, the result is 0.

VEC_ALL_LT(ARG1, ARG2)

Purpose

Tests whether all elements of the first argument are less than the corresponding elements of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` integer, unsigned, or real vector

ARG2
    An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if all elements of `ARG1` are less than the corresponding elements of `ARG2`. Otherwise, the result is 0.

VEC_ALL_NAN(ARG1)

Purpose

Tests whether each element of the given vector is a `NaN`.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` real vector

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is a NaN. Otherwise, the result is 0.

VEC_ALL_NE(ARG1, ARG2)

Purpose

Tests whether all sets of corresponding elements of the given vectors are not equal.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NGE(ARG1, ARG2)

Purpose

Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NGT(ARG1, ARG2)

Purpose

Tests whether each element of the first argument is not greater than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is not greater than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NLE(ARG1, ARG2)

Purpose

Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is not less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NLT(ARG1, ARG2)

Purpose

Tests whether each element of the first argument is not less than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ALL_NUMERIC(ARG1)

Purpose

Tests whether each element of the given vector is numeric (not a NaN).

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) real vector

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if each element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.

VEC_AND(ARG1, ARG2)

Purpose

Performs a bitwise AND of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the bitwise AND of ARG1 and ARG2.

VEC_ANDC(ARG1, ARG2)

Purpose

Performs a bitwise AND of the first argument and the bitwise complement of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the bitwise AND of ARG1 with the bitwise complement of ARG2.

VEC_ANY_EQ(ARG1, ARG2)

Purpose

Tests whether any set of corresponding elements of the given vectors are equal.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` vector

ARG2
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if any element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_GE(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer, unsigned, or real vector

ARG2
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if any element of ARG1 is greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_GT(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is greater than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is greater than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_LE(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_LT(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is less than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is less than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NAN(ARG1)

Purpose

Tests whether any element of the given vector is a NaN.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is a NaN. Otherwise, the result is 0.

VEC_ANY_NE(ARG1, ARG2)

Purpose

Tests whether any set of corresponding elements of the given vectors are not equal.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NGE(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NGT(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is not greater than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
   An `INTENT(IN)` real vector

ARG2
   An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if any element of `ARG1` is not greater than the corresponding element of `ARG2`. Otherwise, the result is 0.

VEC_ANY_NLE(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
   An `INTENT(IN)` real vector

ARG2
   An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if any element of `ARG1` is not less than or equal to the corresponding element of `ARG2`. Otherwise, the result is 0.

VEC_ANY_NLT(ARG1, ARG2)

Purpose

Tests whether any element of the first argument is not less than the corresponding element of the second argument.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) real vector

ARG2
   An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.

VEC_ANY_NUMERIC(ARG1)

Purpose

Tests whether any element of the given vector is numeric (not a NaN).

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) real vector

Result type and attributes

The result is an INTEGER(4).

Result value

The result is 1 if any element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.

VEC_ANY_OUT(ARG1, ARG2)

Purpose

Tests whether the value of any element of a given vector is outside of a given range.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` `REAL(4)` vector

ARG2
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is an `INTEGER(4)`.

Result value

The result is 1 if both of the following conditions are satisfied; otherwise, the result is 0.

- All the elements of ARG2 have a value greater than or equal to 0.
- The absolute value of any element of ARG1 is greater than the value of the corresponding element of ARG2 or less than the negative of the value of the corresponding element of ARG2.

VEC_AVG(ARG1, ARG2)

Purpose

Returns a vector containing the rounded average of each set of corresponding elements of two given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer or unsigned vector whose elements are of kind 1, 2 or, 4

ARG2
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is the rounded average of the values of the corresponding elements of ARG1 and ARG2.

VEC_BPERM (ARG1, ARG2)

Purpose

Gathers up to 16 1-bit values from a quadword or from each doubleword element in the specified order, and places them in the specified order either in the rightmost 16 bits of the leftmost doubleword of the result vector register or in the rightmost 8 bits of each doubleword of the result vector register according to the

element types, with the rest of the result set to 0.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) UNSIGNED(1) or UNSIGNED(8) vector

ARG2
An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

The following table describes the types of the result and the function arguments.

Table 61. Types of the returned value and function arguemnts

result	ARG1	ARG2
UNSIGNED(1) vector	An INTENT(IN) UNSIGNED(1) vector[1]	An INTENT(IN) UNSIGNED(1) vector[1]
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(8) vector[2]	An INTENT(IN) UNSIGNED(1) vector[2]

Note:

1. This combination of data types of the arguments of the intrinsic procedure is valid only when -qarch(-mcpu) is set to target POWER8 or POWER9 processors.
2. This combination of data types of the arguments of the intrinsic procedure is valid only when -qarch(-mcpu) is set to target POWER9 processors.

Result value

- When the data type of ARG1 is UNSIGNED(1) vector, which is valid only when -qarch is set to target POWER8 or POWER9:

  Suppose $i(0<=i<16)$ and j. Let i denote the element index of ARG2, and let j denote the byte value of element i of ARG2:

  – If $j>=128$, bit $48+i$ of doubleword 0 is set to 0.

  – If $j<128$, bit $48+i$ of the result is set to the value of bit j of ARG1.

  – All other bits are set to 0.

- When the data type of ARG1 is UNSIGNED(8) vector, which is valid only when -qarch is set to target POWER9:

  Suppose $i(0<=i<2)$, $j(0<=j<8)$ and k. Let i denote the doubleword element index of ARG1; let j denote the element index of ARG2; and let k denote the byte value of element j of ARG2:

  – If $k>=64$, bit $56+j$ of element i is set to 0.

  – If $k<64$, bit $56+j$ of element i is set to the value of bit k of element i of ARG1.

  – All other bits are set to 0.

VEC_CEIL(ARG1)

Purpose

Returns a vector containing the smallest representable floating-point integral values greater than or equal to the values of the corresponding elements of the given vector.

Note: `VEC_CEIL` is another name for `VEC_ROUNDP`. For details, see "VEC_ROUNDP(ARG1)" on page 847.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` real vector

Result type and attributes

The result is a vector of the same type as `ARG1`.

Result value

Each element of the result contains the smallest representable floating-point integral value greater than or equal to the value of the corresponding element of `ARG1`.

VEC_CIPHER_BE(ARG1,ARG2)

Purpose

Performs one round of the AES cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state `ARG1` by using a given round key `ARG2`.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1) vector

ARG2
    An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

An UNSIGNED(1) vector

Result value

Returns the resulting intermediate state.

VEC_CIPHERLAST_BE(ARG1,ARG2)

Purpose

Performs the final round of the AES cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state ARG1 by using a given round key ARG2.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1) vector

ARG2
    An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

An UNSIGNED(1) vector

Result value

Returns the resulting final state.

VEC_CMPB(ARG1, ARG2)

Purpose

Performs a bounds comparison of each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) REAL(4) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is an INTEGER(4) vector.

Result value

Each element of the result has the value 0 if both of the following conditions are satisfied:

- All the elements of ARG2 have a value greater than or equal to 0.
- All the elements of ARG1 have a value less than or equal to the value of the corresponding element of ARG2 and greater than or equal to the negative of the value of the corresponding element of ARG2.

Otherwise, the result is determined as follows:

- If an element of ARG2 is greater than or equal to zero, the following rules comply:
  - If the absolute value of the corresponding element of ARG1 is equal to the value of the corresponding element of ARG2, the value of the corresponding element of the result is 0.
  - If the absolute value of the corresponding element of ARG1 is greater than the value of the corresponding element of ARG2, the value of the corresponding element of the result is negative.
  - If the absolute value of the corresponding element of ARG1 is less than the value of the corresponding element of ARG2, the value of the corresponding element of the result is positive.
- If an element of ARG2 is less than zero, the value of the element of the result is positive if the value of the corresponding element of ARG1 is less than or equal to the value of the element of ARG2, and negative otherwise.

VEC_CMPEQ(ARG1, ARG2)

Purpose

Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table lists the result types for VEC_CMPEQ().

Table 62. Result types for VEC_CMPEQ()

ARG1	Result type
VECTOR(INTEGER(1))	VECTOR(UNSIGNED(1))
VECTOR(UNSIGNED(1))	

Table 62. Result types for `VEC_CMPEQ()` (continued)

ARG1	Result type
VECTOR(INTEGER(2))	VECTOR(UNSIGNED(2))
VECTOR(UNSIGNED(2))	
VECTOR(INTEGER(4))	VECTOR(UNSIGNED(4))
VECTOR(UNSIGNED(4))	
VECTOR(REAL(4))	
VECTOR(INTEGER(8))	VECTOR(UNSIGNED(8))
VECTOR(UNSIGNED(8))	
VECTOR(REAL(8))	

Result value

For each element of the result, the value of each bit is 1 if the corresponding elements of ARG1 and ARG2 are equal. Otherwise, the value of each bit is 0.

VEC_CMPGE(ARG1, ARG2)

Purpose

Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
　　An INTENT(IN) VECTOR(INTEGER(8)), VECTOR(UNSIGNED(8)), or real vector.

ARG2
　　An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

The following table describes the types of the result.

Table 63. Types of the result and function arguments

result	ARG1	ARG2
UNSIGNED(8) vector	An INTENT(IN) INTEGER(8) vector	An INTENT(IN) INTEGER(8) vector
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(8) vector
UNSIGNED(4) vector	An INTENT(IN) REAL(4) vector	An INTENT(IN) REAL(4) vector
UNSIGNED(8) vector	An INTENT(IN) REAL(8) vector	An INTENT(IN) REAL(8) vector

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CMPGT(ARG1, ARG2)

Purpose

Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` integer, unsigned, or real vector

ARG2
    An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The following table lists the result types for `VEC_CMPGT()`.

Table 64. Result types for `VEC_CMPGT()`

ARG1	Result type
VECTOR(INTEGER(1))	VECTOR(UNSIGNED(1))
VECTOR(UNSIGNED(1))	
VECTOR(INTEGER(2))	VECTOR(UNSIGNED(2))
VECTOR(UNSIGNED(2))	
VECTOR(INTEGER(4))	VECTOR(UNSIGNED(4))
VECTOR(UNSIGNED(4))	
VECTOR(REAL(4))	
VECTOR(INTEGER(8))	VECTOR(UNSIGNED(8))
VECTOR(UNSIGNED(8))	
VECTOR(REAL(8))	

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CMPLE(ARG1, ARG2)

Purpose

Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) VECTOR(INTEGER(8)), VECTOR(UNSIGNED(8)), or real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

The following table describes the types of the result.

Table 65. Types of the result and function arguments

result	ARG1	ARG2
UNSIGNED(8) vector	An INTENT(IN) INTEGER(8) vector	An INTENT(IN) INTEGER(8) vector
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(8) vector
UNSIGNED(4) vector	An INTENT(IN) REAL(4) vector	An INTENT(IN) REAL(4) vector
UNSIGNED(8) vector	An INTENT(IN) REAL(8) vector	An INTENT(IN) REAL(8) vector

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CMPLT(ARG1, ARG2)

Purpose

Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table lists the result types for VEC_CMPLT().

Table 66. Result types for VEC_CMPLT()

ARG1	Result type
VECTOR(INTEGER(1))	VECTOR(UNSIGNED(1))
VECTOR(UNSIGNED(1))	
VECTOR(INTEGER(2))	VECTOR(UNSIGNED(2))
VECTOR(UNSIGNED(2))	
VECTOR(INTEGER(4))	VECTOR(UNSIGNED(4))
VECTOR(UNSIGNED(4))	
VECTOR(REAL(4))	
VECTOR(INTEGER(8))	VECTOR(UNSIGNED(8))
VECTOR(UNSIGNED(8))	
VECTOR(REAL(8))	

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.

VEC_CMPNE(ARG1, ARG2)

Purpose

Returns a vector containing the results of comparing each set of the corresponding elements of the given vectors for inequality.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) vector whose elements are of the following types:
- integer of kind 1, 2, 4 or 8
- unsigned of kind 1, 2, 4 or 8
- real of kind 4 or 8

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table describes the type of the result.

Table 67. Types of the returned value and function arguments

result	ARG1	ARG2
UNSIGNED(1) vector	An INTENT(IN) INTEGER(1) vector	the same type as ARG1
	An INTENT(IN) UNSIGNED(1) vector	
UNSIGNED(2) vector	An INTENT(IN) INTEGER(2) vector	
	An INTENT(IN) UNSIGNED(2) vector	
UNSIGNED(4) vector	An INTENT(IN) INTEGER(4) vector	
	An INTENT(IN) UNSIGNED(4) vector	
	An INTENT(IN) REAL(4) vector	
UNSIGNED(8) vector	An INTENT(IN) INTEGER(8) vector	
	An INTENT(IN) UNSIGNED(8) vector	
	An INTENT(IN) REAL(8) vector	

Result value

For each element of the result, the value of each bit is 1 if the corresponding elements of ARG1 and ARG2 are not equal; otherwise, the value is 0.

VEC_CMPNEZ(ARG1, ARG2)

Purpose

Returns a vector that contains the results of comparing each set of the corresponding elements of the given vectors for inequality, or the results of testing the corresponding element of given vectors for the value of zero.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, or 4

ARG2
　　An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table describes the type of the result.

Table 68. Types of the returned value and function arguments

result	ARG1	ARG2
UNSIGNED(1) vector	An INTENT(IN) INTEGER(1) vector	the same type as ARG1
	An INTENT(IN) UNSIGNED(1) vector	
UNSIGNED(2) vector	An INTENT(IN) INTEGER(2) vector	
	An INTENT(IN) UNSIGNED(2) vector	
UNSIGNED(4) vector	An INTENT(IN) INTEGER(4) vector	
	An INTENT(IN) UNSIGNED(4) vector	

Result value

For each element of the result, the value of each bit is 1 in one of the following cases; otherwise, the value of each bit is 0.
- the corresponding elements of ARG1 and ARG2 are not equal.
- the corresponding element of ARG1 or ARG2 is 0.

VEC_CNTLZ(ARG1)

Purpose

Counts the most significant zero bits of each element of the given vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
　　An INTENT(IN) integer or unsigned vector.

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is set to the number of leading zeros of the corresponding element of ARG1.

VEC_CNTLZ_LSBB(ARG1)

Purpose

Counts the leading byte elements of the given vector that have a least significant bit of 0.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1) or UNSIGNED(1) vector

Result type and attributes

The result has a data type of INTEGER(4).

Result value

The result is set to the number of leading byte elements of ARG1 that have a least significant bit of 0.

VEC_CNTTZ(ARG1)

Purpose

Counts the least significant zero bits of each element of the given vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, 4, or 8

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is set to the number of trailing zeros of the corresponding element of ARG1.

VEC_CNTTZ_LSBB(ARG1)

Purpose

Counts the trailing byte elements of the given vector that have a least significant bit of 0.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `INTEGER(1)` or `UNSIGNED(1)` vector

Result type and attributes

The result has a data type of `INTEGER(4)`.

Result value

The result is set to the number of trailing byte elements of ARG1 that have a least significant bit of 0.

VEC_CONCAT(ARG1, ARG2)

Purpose

Concatenates two elements to form a vector.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Pure function

Argument type and attributes

ARG1
    An `INTENT(IN)` `INTEGER(8)` or `REAL(8)`

ARG2
    An `INTENT(IN)` scalar of the same type as ARG1

Result type and attributes

The result is a vector whose elements are of the same type as ARG1.

Result value

The resulting vector consists of the two scalar elements, `ARG1` and `ARG2`, assigned to elements 0 and 1, respectively.

VEC_CONVERT(V, MOLD)

Purpose

Converts a vector to a vector of a given type.

Class

Pure function

Argument type and attributes

V   An `INTENT(IN)` vector

MOLD
   An `INTENT(IN)` vector

   If it is a variable, it need not be defined.

Result type and attributes

The result is a vector of the same type as `MOLD`.

Result value

The result is as if it were on the left-hand side of an intrinsic assignment with `V` on the right-hand side.

VEC_CPSGN(ARG1, ARG2)

Purpose

Returns a vector by copying the sign of the elements in vector `ARG1` to the sign of the corresponding elements in vector `ARG2`.

Class

Elemental function

Argument type and attributes

ARG1
   An `INTENT(IN)` real vector

ARG2
   An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is of the same type as `ARG1`.

VEC_CTD(ARG1, ARG2)

Purpose

Converts an integer vector into a `REAL(8)` vector.

Result value

For each element of the result, converts the corresponding element of ARG1 to `REAL(8)`, and divides the result by 2 to the power of ARG2.

Note: The second and fourth elements of ARG1 are not used when ARG1 is an `INTEGER(4)` or `UNSIGNED(4)` vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer or unsigned vector whose elements are of kind 4 or 8

ARG2
An `INTENT(IN)` integer. Its value must be specified by a constant expression with a value in the range 0 - 31 inclusive

Result type and attributes

The result is a `REAL(8)` vector.

VEC_CTF(ARG1, ARG2)

Purpose

Converts an integer vector into a `REAL(4)` vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer or unsigned vector whose elements are of kind 4 or 8

ARG2
An `INTENT(IN)` integer. Its value must be specified by a constant expression whose value is in the range 0 and 31 inclusive

Result type and attributes

The result is a `REAL(4)` vector.

Result value

The value of each element of the result is the closest floating-point estimate of the value of the corresponding element of ARG1 divided by 2 to the power of ARG2.

Note: The second and fourth elements of the result vector are undefined when the elements of ARG1 are of kind 8.

VEC_CTS(ARG1, ARG2)

Purpose

Converts a real vector into an `INTEGER(4)` vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

ARG2
An `INTENT(IN)` integer. Its value must be specified by a constant expression whose value is in the range 0 to 31 inclusive.

Result type and attributes

The result is an `INTEGER(4)` vector.

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2.

Note: The second and fourth elements of the result vector are undefined when ARG1 is a `REAL(8)` vector.

VEC_CTSL(ARG1, ARG2)

Purpose

Converts a real vector into an `INTEGER(8)` vector.

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2 and converting the result into an integer.

Note: This function does not use the second and fourth elements of ARG1 when ARG1 is a `REAL(4)` vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

ARG2
　　An `INTENT(IN)` integer. Its value must be specified by a constant expression
　　with a value in the range 0 - 31 inclusive.

Result type and attributes

The result is an `INTEGER(8)` vector.

VEC_CTU(ARG1, ARG2)

Purpose

Converts a real vector into an `UNSIGNED(4)` vector.

Class

Elemental function

Argument type and attributes

ARG1
　　An `INTENT(IN)` real vector

ARG2
　　An `INTENT(IN)` integer. Its value must be specified by a constant expression
　　whose value is in the range 0 to 31 inclusive.

Result type and attributes

The result is an `UNSIGNED(4)` vector.

Result value

The value of each element of the result is the saturated value obtained by
multiplying the corresponding element of ARG1 by 2 to the power of ARG2.

Note: The second and fourth elements of the result vector are undefined when
ARG1 is a `REAL(8)` vector.

VEC_CTUL(ARG1, ARG2)

Purpose

Converts a real vector into an `UNSIGNED(8)` vector.

Result value

The value of each element of the result is the saturated value that is obtained by
multiplying the corresponding element of ARG1 by 2 to the power of ARG2.

Note: The second and fourth elements of the result vector are undefined when
ARG1 is a `REAL(8)` vector.

Class

Elemental function

Argument type and attributes

ARG1
> An `INTENT(IN)` real vector.

ARG2
> An `INTENT(IN)` integer. Its value must be specified by a constant expression with a value in the range 0 - 31 inclusive.

Result type and attributes

The result is an `UNSIGNED(8)` vector.

VEC_CVF(ARG1)

Purpose

Converts a `REAL(4)` vector into a `REAL(8)` vector, and vice versa.

Class

Elemental function

Argument type and attributes

ARG1
> An `INTENT(IN)` real vector

Result type and attributes

The following table describes the types of the result.

Table 69. Types of the returned value and function argument

result	ARG1
`REAL(4)` vector	An `INTENT(IN)` `REAL(8)` vector
`REAL(8)` vector	An `INTENT(IN)` `REAL(4)` vector

Result Value

When ARG1 is a `REAL(4)` vector, each element of the result is equal to the corresponding element of ARG1, converted to `REAL(8)`. The second and fourth elements of ARG1 are not used.

When ARG1 is a `REAL(8)` vector, the first and third elements of the result are equal to the corresponding elements of ARG1, converted to `REAL(4)`. The values of the second and fourth elements of the result vector are undefined.

VEC_DIV(ARG1, ARG2)

Purpose

Divides the elements in vector ARG1 by the corresponding elements in vector ARG2 and then assigns the result to the corresponding elements in the result vector.

Note: For integer and unsigned vectors, this function emulates the operation.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

VEC_DSS(ARG1)

Purpose

Stops the data stream read specified by ARG1.

Class

Subroutine

Argument type and attributes

ARG1
An INTENT(IN) integer and must be specified by a constant expression whose value is in the range 0 through 3 inclusive.

VEC_DSSALL

Purpose

Stops all data stream reads.

Class

Subroutine

VEC_DST(ARG1, ARG2, ARG3)

Purpose

Initiates the data read of a line into cache in a state most efficient for reading.

The data stream specified by ARG3 is read beginning at the address specified by ARG1 using the control word specified by ARG2. After you use this intrinsic procedure, the specified data stream is relatively persistent.

Class

Subroutine

Argument type and attributes

ARG1

An INTENT(IN) variable of any of the following types:
- INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) integer

ARG3

An INTENT(IN) integer

It must be specified by a constant expression whose value is in the range 0 - 3 inclusive.

VEC_DSTST(ARG1, ARG2, ARG3)

Purpose

Initiates the data read of a line into cache in a state most efficient for writing.

The data stream specified by ARG3 is read beginning at the address specified by ARG1 using the control word specified by ARG2. Use of this intrinsic procedure indicates that the specified data stream is relatively persistent in nature.

Class

Subroutine

Argument type and attributes

ARG1

An INTENT(IN) variable of any of the following types:
- INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) integer

ARG3

An INTENT(IN) integer

It must be specified by a constant expression whose value is in the range 0 - 3 inclusive.

VEC_DSTSTT(ARG1, ARG2, ARG3)

Purpose

Initiates the data read of a line into cache in a state most efficient for writing.

The data stream specified by ARG3 is read beginning at the address specified by ARG1 using the control word specified by ARG2. Use of this intrinsic procedure indicates that the specified data stream is relatively transient in nature.

Class

Subroutine

Argument type and attributes

ARG1
An INTENT(IN) variable of any of the following types:
- INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2
An INTENT(IN) integer

ARG3
An INTENT(IN) integer

It must be specified by a constant expression whose value is in the range 0 - 3 inclusive.

VEC_DSTT(ARG1, ARG2, ARG3)

Purpose

Initiates the data read of a line into cache in a state most efficient for reading.

The data stream specified by ARG3 is read beginning at the address specified by ARG1 using the control word specified by ARG2. Use of this intrinsic procedure indicates that the specified data stream is relatively transient in nature.

Class

Subroutine

Argument type and attributes

ARG1
An INTENT(IN) variable of any of the following types:
- INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2
An INTENT(IN) integer

ARG3
An INTENT(IN) integer

It must be specified by a constant expression whose value is in the range 0 - 3 inclusive.

VEC_EQV(ARG1, ARG2)

Purpose

Performs a bitwise equivalence operation on the input vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

The result is of the same type as ARG1.

Result value

Each bit of the result is set to the result of the bitwise operation (ARG1 == ARG2) of the corresponding bits of ARG1 and ARG2. For 0 <= i < 128, bit i of the result is set to 1 only if bit i of ARG1 is equal to bit i of ARG2.

VEC_EXPTE(ARG1)

Purpose

Returns a vector containing estimates of 2 raised to the values of the corresponding elements of a given vector.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) REAL(4) vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the estimated value of 2 raised to the value of the corresponding element of ARG1.

VEC_EXTRACT(ARG1, ARG2)

Purpose

Returns the value of element ARG2 from the vector ARG1.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer or real vector

ARG2
    An INTENT(IN) integer

Result type and attributes

The result has the same type and kind as the elements of ARG1.

Result value

This function uses the modulo arithmetic on ARG2 to determine the element number. For example, if ARG2 is out of range, the compiler uses ARG2 modulo the number of elements in ARG1 to determine the element position.

VEC_EXTRACT_EXP(ARG1)

Purpose

Returns a vector that contains the exponent of the given vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

Result type and attributes

The following table describes the type of the result.

Table 70. Types of the returned value and function argument

result	ARG1
UNSIGNED(4) vector	An INTENT(IN) REAL(4) vector
UNSIGNED(8) vector	An INTENT(IN) REAL(8) vector

Result value

Each element of the result is extracted from the exponent field of the corresponding element of ARG1.

Note: The extracted exponent of ARG1 is treated as a biased exponent in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.

VEC_EXTRACT_SIG(ARG1)

Purpose

Returns a vector that contains the significand of the given vector.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

Result type and attributes

The following table describes the type of the result.

Table 71. Types of the returned value and function argument

result	ARG1
`UNSIGNED(4)` vector	An `INTENT(IN) REAL(4)` vector
`UNSIGNED(8)` vector	An `INTENT(IN) REAL(8)` vector

Result value

Each element of the result is extracted from the significand field of the corresponding element of ARG1.

Note:
* The extracted significand of ARG1 is in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.
* The value of the implicit leading digit that is included in the result is not encoded in the *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, but implied by the exponent.

VEC_FIRST_MATCH_INDEX(ARG1, ARG2)

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of equality.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4), UNSIGNED(1), UNSIGNED(2) or UNSIGNED(4) vector.

ARG2
An INTENT(IN) of the same type as ARG1.

Result type and attributes

The data type of the result is UNSIGNED(4).

Result value

Returns the byte index of the position of the first equal element between ARG1 and ARG2. If all the corresponding elements are unequal between ARG1 and ARG2, returns the number of the elements of either ARG1 or ARG2.

VEC_FIRST_MATCH_OR_EOS_INDEX(ARG1, ARG2)

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of equality or the position of the end-of-string terminator \0.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4), UNSIGNED(1), UNSIGNED(2) or UNSIGNED(4) vector.

ARG2
An INTENT(IN) of the same type as ARG1.

Result type and attributes

The data type of the result is `UNSIGNED(4)`.

Result value

Returns the byte index of the position of the first equal element between `ARG1` and `ARG2` or the position of an end-of-string terminator, *\0*.

If both of the following conditions are met,
- There is no equality between the corresponding elements of `ARG1` and `ARG2`;
- There is no end-of-string terminator;

returns the number of the elements of either `ARG1` or `ARG2`.

VEC_FIRST_MISMATCH_INDEX(ARG1, ARG2)

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of inequality.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, `UNSIGNED(1)`, `UNSIGNED(2)` or `UNSIGNED(4)` vector.

ARG2
    An `INTENT(IN)` of the same type as `ARG1`.

Result type and attributes

The data type of the result is `UNSIGNED(4)`.

Result value

Returns the byte index of the position of the first unequal element between `ARG1` and `ARG2`. If all the corresponding elements are equal between `ARG1` and `ARG2`, returns the number of the elements of either `ARG1` or `ARG2`.

VEC_FIRST_MISMATCH_OR_EOS_INDEX(ARG1, ARG2)

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of inequality or the position of the end-of-string terminator *\0*.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4), UNSIGNED(1), UNSIGNED(2) or UNSIGNED(4) vector.

ARG2
An INTENT(IN) of the same type as ARG1.

Result type and attributes

The data type of the result is UNSIGNED(4).

Result value

Returns the byte index of the first position of the unequal element between ARG1 and ARG2 or the position of an end-of-string terminator, \0.

If both of the following conditions are met,
- There is no unequality between the corresponding elements of ARG1 and ARG2;
- There is no end-of-string terminator;

returns the number of the elements of either ARG1 or ARG2.

VEC_FLOOR(ARG1)

Purpose

Returns a vector containing the largest representable floating-point integral values less than or equal to the values of the corresponding elements of the given vector.

Note: `VEC_FLOOR` is another name for `VEC_ROUNDM`. For details, see "VEC_ROUNDM(ARG1)" on page 847.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the largest representable floating-point integral value less than or equal to the value of the corresponding element of ARG1.

VEC_GBB(ARG1)

Purpose

Performs a gather-bits-by-bytes operation on the input.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) VECTOR(INTEGER(8)) or VECTOR(UNSIGNED(8)).

Result type and attributes

The result is of the same type as ARG1.

Result value

Each doubleword element of the result is set as follows: Let x(i) (0 <= i < 8) denote the byte elements of the corresponding input doubleword element, with x(7) the most significant byte. For each pair of i and j (0 <= i < 8, 0 <= j < 8), the jth bit of the ith byte element of the result is set to the value of the ith bit of the jth byte element of the input.

VEC_INSERT(ARG1, ARG2, ARG3)

Purpose

Returns a copy of the vector ARG2 with the value of its element ARG3 replaced by ARG1.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) integer, REAL(4) or REAL(8)

ARG2
  An INTENT(IN) vector whose elements have the same type as ARG1

ARG3
  An INTENT(IN) integer

Result type and attributes

The result is of the same type as ARG2.

Result value

This function uses the modulo arithmetic on `ARG3` to determine the element
number. For example, if `ARG3` is out of range, the compiler uses `ARG3` modulo the
number of elements in the vector to determine the element position.

VEC_INSERT_EXP(ARG1, ARG2)

Purpose

Returns a vector that combines the exponents of elements from one vector with the
signs and the significands of elements from another vector.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9
technology.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` unsigned or real vector whose elements are of kind 4 or 8

ARG2
    An `INTENT(IN)`

    If `ARG1` is an unsigned or real vector whose elements are of kind 4, `ARG2` is an
    `UNSIGNED(4)` vector. If `ARG1` is an unsigned or real vector whose elements are of
    kind 8, `ARG2` is an `UNSIGNED(8)` vector.

Result type and attributes

The following table describes the type of the result.

Table 72. Types of the returned value and function arguments

result	ARG1	ARG2
`REAL(4)` vector	An `INTENT(IN)` `UNSIGNED(4)` vector	An `INTENT(IN)` `UNSIGNED(4)` vector
`REAL(4)` vector	An `INTENT(IN)` `REAL(4)` vector	An `INTENT(IN)` `UNSIGNED(4)` vector
`REAL(8)` vector	An `INTENT(IN)` `UNSIGNED(8)` vector	An `INTENT(IN)` `UNSIGNED(8)` vector
`REAL(8)` vector	An `INTENT(IN)` `REAL(8)` vector	An `INTENT(IN)` `UNSIGNED(8)` vector

Result value

Each element of the result is generated by combining the exponent of the
corresponding element of `ARG2` with the sign and the significand of the
corresponding element of `ARG1`.

Note: The inserted exponent of ARG2 is treated as a biased exponent in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.

VEC_LD(ARG1, ARG2)

Purpose

Loads a vector from the given memory address.

Class

Function

Argument type and attributes

ARG1
An INTENT(IN) integer

ARG2
An INTENT(IN) variable. The variable can be any of the following types:
- INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

Result type and attributes

If ARG2 is a vector, then the result is of the same type as ARG2. Otherwise, the result is a vector containing elements of the same type as ARG2.

Result value

The value of ARG1 is added to the address of ARG2, and the sum is truncated to a multiple of 16 bytes. The result is the content of the 16 bytes of memory starting at this address.

VEC_LDE(ARG1, ARG2)

Purpose

Loads an element from a given memory address into a vector.

Class

Function

Argument type and attributes

ARG1
An INTENT(IN) integer

ARG2
An INTENT(IN) variable of type INTEGER(1), INTEGER(2), INTEGER(4), or REAL(4)

Result type and attributes

The result is a vector containing elements of the same type as ARG2.

Result value

The effective address is the sum of ARG1 and the address specified by ARG2, truncated to a multiple of the size in bytes of an element of the result vector. The contents of memory at the effective address are loaded into the result vector at the byte offset corresponding to the four least significant bits of the effective address. The remaining elements of the result vector are undefined.

Related information:

 -qaltivec

 Vector element order toggling

VEC_LDL(ARG1, ARG2)

Purpose

Loads a vector from a given memory address, and marks the cache line containing the data as Least Recently Used.

Class

Function

Argument type and attributes

ARG1
An `INTENT(IN)` integer

ARG2
An `INTENT(IN)` variable

The variable can be any of the following types:
- `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, or `REAL(4)`
- `VECTOR(INTEGER(1))`, `VECTOR(INTEGER(2))`, or `VECTOR(INTEGER(4))`
- `VECTOR(UNSIGNED(1))`, `VECTOR(UNSIGNED(2))`, or `VECTOR(UNSIGNED(4))`
- `VECTOR(REAL(4))`
- `VECTOR(PIXEL)`

Result type and attributes

If `ARG2` is a vector, the result is of the same type as `ARG2`. Otherwise, the result is a vector containing elements of the same type as `ARG2`.

Result value

`ARG1` is added to the address specified by `ARG2`, and the sum is truncated to a multiple of 16 bytes. The result is the contents of the 16 bytes of memory starting at this address. This data is marked as Least Recently Used.

Related information:

 -qaltivec

Vector element order toggling

VEC_LOGE(ARG1)

Purpose

Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `REAL(4)` vector

Result type and attributes

The result is a vector of the same type as `ARG1`.

Result value

Each element of the result contains the estimated value of the base-2 logarithm of the corresponding element of `ARG1`.

VEC_LOAD_SPLATS(ARG1, ARG2)

Purpose

Loads a 4-byte element from the memory address specified by the displacement `ARG1` and the address provided by `ARG2` and then splats it onto a vector.

This intrinsic procedure is valid only when `-qarch` is set to utilize POWER9 technology.

Argument type and attributes

ARG1
    An `INTENT(IN)` `INTEGER(8)`

ARG2
    An `INTENT(IN)` `INTEGER(4)`, or `REAL(4)`

Result type and attributes

The following table describes the types of the result.

Table 73. Result and argument types

result	ARG1	ARG2
`INTEGER(4)` vector	An `INTENT(IN)` `INTEGER(8)`	`INTEGER(4)`
`REAL(4)` vector	An `INTENT(IN)` `INTEGER(8)`	`REAL(4)`

Result value

The result is a vector with each element set to the 4-byte element from the effective address calculated by adding the displacement provided by ARG1 and the address provided by ARG2. The effective address is not truncated to a multiple of 16 bytes.

VEC_LVSL(ARG1, ARG2)

Purpose

Returns a vector useful for aligning non-aligned data.

Class

Function

Argument type and attributes

ARG1
An `INTENT(IN)` integer

ARG2
An `INTENT(IN)` variable of type `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, or `REAL(4)`

Result type and attributes

The result is an `UNSIGNED(1)` vector.

Result value

The first element of the result vector is the sum of the value of ARG1 and the address of ARG2, modulo 16. The value of each successive element is the value of the previous element plus 1.

VEC_LVSR(ARG1, ARG2)

Purpose

Returns a vector useful for aligning non-aligned data.

Class

Function

Argument type and attributes

ARG1
An `INTENT(IN)` integer

ARG2
An `INTENT(IN)` variable of type `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, or `REAL(4)`

Result type and attributes

The result is an `UNSIGNED(1)` vector.

Result value

The effective address is the sum of the value of ARG1 and the address of ARG2, modulo 16. The first element of the result vector contains the value 16 minus the effective address. The value of each successive element is the value of the previous element plus 1.

VEC_MADD(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a fused multiply-add operation on each corresponding set of elements of three given vectors.

Class

Elemental function

Argument type and attributes

ARG1
> An INTENT(IN) real vector

ARG2
> An INTENT(IN) vector of the same type as ARG1

ARG3
> An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the product of the values of the corresponding elements of ARG1 and ARG2, added to the value of the corresponding element of ARG3.

VEC_MADDS(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a saturated multiply-high-and-add operation on each corresponding set of elements of three given vectors.

Class

Elemental function

Argument type and attributes

ARG1
> An INTENT(IN) INTEGER(2) vector

ARG2
> An INTENT(IN) vector of the same type as ARG1

ARG3
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

For each element of the result, the value is produced in the following way: the values of the corresponding elements of ARG1 and ARG2 are multiplied. The value of the 17 most significant bits of this product is then added, using 16-bit-saturated addition, to the value of the corresponding element of ARG3.

VEC_MAX(ARG1, ARG2)

Purpose

Returns a vector that contains the maximum value from each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer vector, unsigned vector, or real vector.

ARG2
An `INTENT(IN)` vector of the same type as ARG1.

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is the maximum of the values of the corresponding elements of ARG1 and ARG2.

VEC_MERGEE(ARG1,ARG2)

Purpose

Merges the values of even-numbered elements of two vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector(integer(4)) or vector(unsigned(4))

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are obtained, in order, from the even-numbered elements of ARG1. The odd-numbered elements of the result are obtained, in order, from the even-numbered elements of ARG2.

Example

```
PROGRAM main
 IMPLICIT NONE

 VECTOR(INTEGER(4)) v1, v2, v3
 INTEGER(4) i4(4), j4(4)

 i4 = [0, 1, 2, 3]
 j4 = [5, 6, 7, 8]
 v1 = VEC_XL(0, i4)
 v2 = VEC_XL(0, j4)
 v3 = VEC_MERGEE(v1,v2)
 CALL VEC_XST(v3, 0, i4)
 PRINT *, 'v3:', i4
END
```

The output is as follows:
```
v3: 0 2 5 7
```
> **Related information**
> "VEC_MERGEO(ARG1,ARG2)" on page 820

VEC_MERGEH(ARG1, ARG2)

Purpose

Merges the most significant halves of two vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are taken, in order, from the high elements of ARG1. The odd-numbered elements of the result are taken, in order, from the high elements of ARG2.

Related reference:

"VEC_MERGEL(ARG1, ARG2)"

Related information:

 -qaltivec

 Vector element order toggling

VEC_MERGEL(ARG1, ARG2)

Purpose

Merges the least significant halves of two vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are taken, in order, from the low elements of ARG1. The odd-numbered elements of the result are taken, in order, from the low elements of ARG2.

Related reference:

"VEC_MERGEH(ARG1, ARG2)" on page 819

Related information:

 -qaltivec

 Vector element order toggling

VEC_MERGEO(ARG1,ARG2)

Purpose

Merges the values of odd-numbered elements of two vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector(integer(4)) or vector(unsigned(4))

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are obtained, in order, from the odd-numbered elements of ARG1. The odd-numbered elements of the result are obtained, in order, from the odd-numbered elements of ARG2.

Example

```
PROGRAM main
 IMPLICIT NONE

 VECTOR(INTEGER(4)) v1, v2, v3
 INTEGER(4) i4(4), j4(4)

 i4 = [0, 1, 2, 3]
 j4 = [5, 6, 7, 8]
 v1 = VEC_XL(0, i4)
 v2 = VEC_XL(0, j4)
 v3 = VEC_MERGEO(v1,v2)
 CALL VEC_XST(v3, 0, i4)
 PRINT *, 'v3:', i4
END
```

The output is as follows:

```
v3: 1 3 6 8
```

Related information
"VEC_MERGEE(ARG1,ARG2)" on page 818

VEC_MFVSCR

Purpose

Copies the contents of the Vector Status and Control Register into the result vector.

Class

Function

Result type and attributes

The result is an UNSIGNED(2) vector.

Result value

The high-order 16 bits of the VSCR are copied into the seventh element of the result. The low-order 16 bits of the VSCR are copied into the eighth element of the result. All other elements are set to zero.

VEC_MIN(ARG1, ARG2)

Purpose

Returns a vector that contains the minimum value from each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer vector, unsigned vector, or real vector.

ARG2
An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is the minimum of the values of the corresponding elements of ARG1 and ARG2.

VEC_MLADD(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a saturated multiply-low-and-add operation on each corresponding set of elements of three given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(2) or UNSIGNED(2) vector

ARG2
An INTENT(IN) INTEGER(2) or UNSIGNED(2) vector

ARG3
An INTENT(IN) vector of the same type as ARG2

Result type and attributes

If `ARG1`, `ARG2`, and `ARG3` are all unsigned vectors, the result is an `UNSIGNED(2)` vector. Otherwise, the result is an `INTEGER(2)` vector.

Result value

The value of each element of the result is the value of the least significant 16 bits of the product of the values of the corresponding elements of `ARG1` and `ARG2`, added to the value of the corresponding element of `ARG3`.

The addition is performed using modular arithmetic.

VEC_MRADDS(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a saturated multiply-high-round-and-add operation for each corresponding set of elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` `INTEGER(2)` vector

ARG2
An `INTENT(IN)` vector of the same type as `ARG1`

ARG3
An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is of the same type as `ARG1`.

Result value

For each element of the result, the value is produced in the following way: the values of the corresponding elements of `ARG1` and `ARG2` are multiplied and rounded such that the 15 least significant bits are 0. The value of the 17 most significant bits of this rounded product is then added, using 16-bit-saturated addition, to the value of the corresponding element of `ARG3`.

VEC_MSUB(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a multiply-subtract operation using the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) real vector

ARG2
   An INTENT(IN) vector of the same type as ARG1

ARG3
   An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

This function multiplies each element in ARG1 by the corresponding element in ARG2 and then subtracts the corresponding element in ARG3 from the result.

VEC_MSUM(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a multiply-sum operation using given vectors.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) integer or unsigned vector whose elements are of kind 1 or 2

ARG2
   An INTENT(IN) vector

   If ARG1 is an INTEGER(1) vector, ARG2 is an UNSIGNED(1) vector. Otherwise, ARG2 is of the same type as ARG1.

ARG3
   An INTENT(IN) vector

   If ARG1 is an integer vector, ARG3 is an INTEGER(4) vector. If ARG1 is an unsigned vector, ARG3 is an UNSIGNED(4) vector.

Result type and attributes

The following table describes the type of the result.

Table 74. Types of the returned value and function arguments

data type of result	data type of ARG1	data type of ARG2	data type of ARG3
INTEGER(4) vector	INTEGER(1) vector	UNSIGNED(1) vector	INTEGER(4) vector

Table 74. Types of the returned value and function arguments (continued)

data type of result	data type of ARG1	data type of ARG2	data type of ARG3
UNSIGNED(4) vector	UNSIGNED(1) vector	UNSIGNED(1) vector	UNSIGNED(4) vector
INTEGER(4) vector	INTEGER(2) vector	INTEGER(2) vector	INTEGER(4) vector
UNSIGNED(4) vector	UNSIGNED(2) vector	UNSIGNED(2) vector	UNSIGNED(4) vector

Result value

For each element n of the result vector, the value is obtained as follows:

- If ARG1 is an INTEGER(1) vector or an UNSIGNED(1) vector, multiply element p of ARG1 by element p of ARG2 where p is from 4n to 4n+3, and then add the sum of these products and element n of ARG3.

- If ARG1 is an INTEGER(2) vector or an UNSIGNED(2) vector, multiply element p of ARG1 by element p of ARG2 where p is from 2n to 2n+1, and then add the sum of these products and element n of ARG3.

All additions are performed by using 32-bit modular arithmetic.

VEC_MSUMS(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a saturated multiply-sum operation using the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(2) or UNSIGNED(2) vector

ARG2
An INTENT(IN) vector of the same type as ARG1

ARG3
An INTENT(IN) vector

If ARG1 is an integer vector, ARG3 is an INTEGER(4) vector. If ARG1 is an unsigned vector, ARG3 is an UNSIGNED(4) vector.

Result type and attributes

The following table describes the type of the result.

Table 75. Types of the returned value and function arguments

data type of result	data type of ARG1	data type of ARG2	data type of ARG3
INTEGER(4) vector	INTEGER(2) vector	INTEGER(2) vector	INTEGER(4) vector
UNSIGNED(4) vector	UNSIGNED(2) vector	UNSIGNED(2) vector	UNSIGNED(4) vector

Result value

For each element n of the result vector, the value is obtained in the following way: multiply element p of ARG1 by element p of ARG2, where p is from 2n to 2n+1; and then add the sum of these products to element n of ARG3. All additions are performed by using 32-bit saturated arithmetic.

VEC_MTVSCR(ARG1)

Purpose

Copies the given value into the Vector Status and Control Register.

The low-order 32 bits of ARG1 are copied into the VSCR.

Class

Subroutine

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4), UNSIGNED(1), UNSIGNED(2), UNSIGNED(4), or PIXEL vector

VEC_MUL(ARG1, ARG2)

Purpose

Returns a vector containing the results of performing a multiply operation using the given vectors.

Note: For integer vectors, this function emulates the operation.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

This function multiplies corresponding elements in the given vectors and then assigns the result to corresponding elements in the result vector.

VEC_MULE(ARG1, ARG2)

Purpose

Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the first element.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, or 4

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table describes the type of the result.

Table 76. Types of the returned value and function arguments

d	a	b
INTEGER(2) vector	An INTENT(IN) INTEGER(1) vector	the same type as ARG1
UNSIGNED(2) vector	An INTENT(IN) UNSIGNED(1) vector	
INTEGER(4) vector	An INTENT(IN) INTEGER(2) vector	
UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(2) vector	
INTEGER(8) vector	An INTENT(IN) INTEGER(4) vector	
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(4) vector	

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n of ARG1 and the value of element 2n of ARG2.

Related information:

-qaltivec

Vector element order toggling

VEC_MULO(ARG1, ARG2)

Purpose

Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the second element.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, or 4

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The following table describes the type of the result.

Table 77. Types of the returned value and function arguments

d	a	b
INTEGER(2) vector	An INTENT(IN) INTEGER(1) vector	the same type as ARG1
UNSIGNED(2) vector	An INTENT(IN) UNSIGNED(1) vector	
INTEGER(4) vector	An INTENT(IN) INTEGER(2) vector	
UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(2) vector	
INTEGER(8) vector	An INTENT(IN) INTEGER(4) vector	
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(4) vector	

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element $2n+1$ of ARG1 and the value of element $2n+1$ of ARG2.

Related information:

-qaltivec

Vector element order toggling

VEC_NABS(ARG1)

Purpose

Returns a vector containing the results of performing a negative-absolute operation using the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` real vector

Result type and attributes

The result is of the same type as `ARG1`.

Result value

This function computes the absolute value of each element in the given vector and then assigns the negated value of the result to the corresponding elements in the result vector.

VEC_NAND(ARG1, ARG2)

Purpose

Performs a bitwise negated-and operation on the input vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` vector

ARG2
    An `INTENT(IN)` vector of the same type as `ARG1`

Result type and attributes

The result is of the same type as `ARG1`.

Result value

Each bit of the result is set to the result of the bitwise operation `NOT(IAND(ARG1, ARG2))` of the corresponding bits of `ARG1` and `ARG2`. For 0 <= i < 128, bit i of the result is set to 0 only if the ith bits of both `ARG1` and `ARG2` are 1.

VEC_NCIPHER_BE(ARG1,ARG2)

Purpose

Performs one round of the AES inverse cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state `ARG1` by using a given round key `ARG2`.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1) vector

ARG2
    An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

An UNSIGNED(1) vector

Result value

Returns the resulting intermediate state.

VEC_NCIPHERLAST_BE(ARG1,ARG2)

Purpose

Performs the final round of the AES inverse cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state ARG1 by using a given round key ARG2.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1) vector

ARG2
    An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

An UNSIGNED(1) vector

Result value

Returns the resulting final state.

VEC_NEARBYINT(ARG1)

Purpose

Returns a vector that contains the rounded values of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` real vector

Result type and attributes

The result is a vector of the same type as `ARG1`.

Result value

Each element of the result contains the value of the corresponding element of `ARG1`, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding. When an input element value is between two integer values, the result value with the larger absolute value is selected.

Related reference:
"VEC_ROUND(ARG1)" on page 846

VEC_NEG(ARG1)

Purpose

Returns a vector containing the negated value of the corresponding elements in the given vector.

Note: For `VECTOR(INTEGER(8))`, this function emulates the operation.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` integer or real vector

Result type and attributes

The result is of the same type as `ARG1`.

Result value

This function multiplies the value of each element in the given vector by -1.0 and then assigns the result to the corresponding elements in the result vector.

VEC_NMADD(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a negative multiply-add operation on the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) real vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

ARG3
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the product of the corresponding elements of ARG1 and ARG2, added to the corresponding elements of ARG3, and then multiplied by -1.0.

VEC_NMSUB(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) real vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

ARG3
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The value of each element of the result is the product of the corresponding elements of ARG1 and ARG2, subtracted from the corresponding element of ARG3.

VEC_NOR(ARG1, ARG2)

Purpose

Performs a bitwise NOR of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) vector

ARG2
  An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the bitwise NOR of ARG1 and ARG2.

VEC_OR(ARG1, ARG2)

Purpose

Performs a bitwise OR of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) vector

ARG2
  An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the bitwise OR of ARG1 and ARG2.

VEC_ORC(ARG1, ARG2)

Purpose

Performs a bitwise OR-with-complement operation of the input vectors.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) integer or unsigned vector.

ARG2

An INTENT(IN) integer or unsigned vector

Result type and attributes

The result is of the same type as ARG1.

Result value

Each bit of the result is set to the result of the bitwise operation IOR(ARG1, NOT(ARG2)) of the corresponding bits of ARG1 and ARG2. For 0 <= i < 128, bit i of the result is set to 1 only if the ith bit of ARG1 is 1 or the ith bit of ARG2 is 0.

VEC_PACK(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) integer or unsigned vector, whose elements are of kind 2, 4 or 8.

ARG2

An INTENT(IN) vector of the same type as ARG1.

Result type and attributes

The following table describes the type of the result.

Table 78. Types of the returned value and function arguments

d	a	b
INTEGER(1) vector	An INTENT(IN) INTEGER(2) vector	the same type as ARG1
UNSIGNED(1) vector	An INTENT(IN) UNSIGNED(2) vector	
INTEGER(2) vector	An INTENT(IN) INTEGER(4) vector	
UNSIGNED(2) vector	An INTENT(IN) UNSIGNED(4) vector	
INTEGER(4) vector	An INTENT(IN) INTEGER(8) vector	
UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(8) vector	

Result value

The value of each element of the result vector is taken from the low-order half of the corresponding element of the result of concatenating ARG1 and ARG2.

Related information:

📄 -qaltivec

📄 Vector element order toggling

VEC_PACKPX(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) UNSIGNED(4) vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a pixel vector.

Result value

The value of each element of the result vector is taken from the corresponding element of the result of concatenating ARG1 and ARG2 in the following way: the least significant bit of the high order byte is stored into the first bit of the result element; the most significant 5 bits of each of the remaining bytes are stored into the remaining portion of the result element.

VEC_PACKS(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector, using saturated values.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector, whose elements are of kind 2, 4 or 8.

ARG2
An `INTENT(IN)` vector of the same type as ARG1.

Result type and attributes

The following table describes the type of the result.

Table 79. Types of the returned value and function arguments

d	a	b
`INTEGER(1)` vector	An `INTENT(IN)` `INTEGER(2)` vector	the same type as ARG1
`UNSIGNED(1)` vector	An `INTENT(IN)` `UNSIGNED(2)` vector	
`INTEGER(2)` vector	An `INTENT(IN)` `INTEGER(4)` vector	
`UNSIGNED(2)` vector	An `INTENT(IN)` `UNSIGNED(4)` vector	
`INTEGER(4)` vector	An `INTENT(IN)` `INTEGER(8)` vector	
`UNSIGNED(4)` vector	An `INTENT(IN)` `UNSIGNED(8)` vector	

Result value

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.

VEC_PACKSU(ARG1, ARG2)

Purpose

Packs information from each element of two vectors into the result vector, using saturated values.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer or unsigned vector, whose elements are of kind 2, 4 or 8.

ARG2
An `INTENT(IN)` vector of the same type as ARG1.

Result type and attributes

The following table describes the type of the result.

Table 80. Types of the returned value and function arguments

d	a	b
UNSIGNED(1) vector	An INTENT(IN) INTEGER(2) vector	the same type as ARG1
UNSIGNED(1) vector	An INTENT(IN) UNSIGNED(2) vector	
UNSIGNED(2) vector	An INTENT(IN) INTEGER(4) vector	
UNSIGNED(2) vector	An INTENT(IN) UNSIGNED(4) vector	
UNSIGNED(4) vector	An INTENT(IN) INTEGER(8) vector	
UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(8) vector	

Result value

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.

VEC_PARITY_LSBB(ARG1)

Purpose

Returns a vector that computes parity on the least significant bit of each byte of each element of the given vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer or unsigned vector whose elements are of kind 4 or 8

Result type and attributes

The following table describes the types of the result.

Table 81. Types of the returned value and function argument

result	ARG1
UNSIGNED(4) vector	INTEGER(4) vector
UNSIGNED(4) vector	UNSIGNED(4) vector
UNSIGNED(8) vector	INTEGER(8) vector

Table 81. Types of the returned value and function argument (continued)

result	ARG1
UNSIGNED(8) vector	UNSIGNED(8) vector

Result value

The value of each element of the result is the parity of the least significant bit of each byte of the corresponding element of ARG1.

VEC_PERM(ARG1, ARG2, ARG3)

Purpose

Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` vector

ARG2
    An `INTENT(IN)` vector of the same type as ARG1

ARG3
    An `INTENT(IN)`, `UNSIGNED(1)` vector

Result type and attributes

The result is of the same type as ARG1.

Result value

Each byte of the result is selected by using the least significant five bits of the corresponding byte of ARG3 as an index into the concatenated bytes of ARG1 and ARG2.

VEC_PERMI(ARG1, ARG2, ARG3)

Purpose

Returns a vector by permuting and combining two eight-byte-long vector elements in ARG1 and ARG2 based on the value of ARG3.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` `INTEGER(8)`, `UNSIGNED(8)`, or `REAL(8)` vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

ARG3
    An integer constant expression with the value of 0, 1, 2, or 3

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of the result vector is based on the value of ARG3.

Table 82. The elements of the result vector

ARG3	First element of result	Second element of result
0	First element of ARG1	First element of ARG2
1	First element of ARG1	Second element of ARG2
2	Second element of ARG1	First element of ARG2
3	Second element of ARG1	Second element of ARG2

VEC_PMSUM_BE(ARG1,ARG2)

Purpose

Performs an exclusive-OR operation by implementing a polynomial addition on each even-odd pair of the polynomial multiplication result of the corresponding elements.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1), UNSIGNED(2), or UNSIGNED(4) vector

ARG2
    An INTENT(IN) vector of the same type and type parameters as ARG1

Result type and attributes

The following table describes the type of the result.

Table 83. Types of the returned value and function arguments

data type of the result	data type of ARG1	data type of ARG2
UNSIGNED(2) vector	An INTENT(IN) UNSIGNED(1) vector	the same type as ARG1
UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(2) vector	
UNSIGNED(8) vector	An INTENT(IN) UNSIGNED(4) vector	

Result value

Each element *i* of the result vector is computed by an exclusive-OR operation of the polynomial multiplication of input elements 2*i of ARG1 and ARG2 and input elements 2*i + 1 of ARG1 and ARG2.

VEC_POPCNT(ARG1)

Purpose

Computes the population count (number of set bits) in each element of the input.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer or unsigned vector.

Result type and attributes

The result is an unsigned vector of the same kind as ARG1.

Result value

Each element of the result is set to the number of set bits in the corresponding element of the input.

VEC_PROMOTE(ARG1, ARG2)

Purpose

Returns a vector with ARG1 in element position ARG2. The values of all the other elements of the constructed vector are undefined.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` `REAL(4)`, `REAL(8)`, or integer vector

ARG2
An `INTENT(IN)` integer

Result type and attributes

The type of the result is a vector whose elements are of the same type as ARG1.

Result value

The result is a vector with ARG1 in element position ARG2. This function uses modulo arithmetic on ARG2 to determine the element number. For example, if ARG2

is out of range, the compiler uses ARG2 modulo the number of elements in the vector to determine the element position. The other elements of the vector are undefined.

VEC_RE(ARG1)

Purpose

Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the estimated value of the reciprocal of the corresponding element of ARG1.

VEC_RECIPDIV(ARG1,ARG2)

Purpose

Returns a vector that contains the division of each elements of ARG1 by the corresponding elements of ARG2, by performing reciprocal estimates and iterative refinement on the elements of ARG2.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) real vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the approximate division of each element of ARG1 by the corresponding element of ARG2. Vector reciprocal estimates and iterative refinement on each element of ARG2 are used to improve the accuracy of the approximation.

Related information

"VEC_RE(ARG1)" on page 841

"VEC_DIV(ARG1, ARG2)" on page 801

VEC_REVB(ARG1)

Purpose

Returns a vector that contains the bytes of the corresponding element of the argument in the reverse byte order.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer, unsigned, or real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the bytes of the corresponding element of ARG1 in the reverse byte order.

Example

```
PROGRAM main

  IMPLICIT NONE

  VECTOR(INTEGER(8)) v1, v2
  INTEGER(8) i8(2)

  i8 = [z'0001020304050607', z'08090a0b0c0d0e0f']
  v1 = VEC_XL(0, i8)
  CALL VEC_XST(v1, 0, i8)
  PRINT 100, 'v1: ', i8
  v2 = VEC_REVB(v1)
  CALL VEC_XST(v2, 0, i8)
  PRINT 100, 'v2: ', i8
  100 FORMAT(A,z16.16,1x,z16.16)

END
```

The output:

```
v1: 0001020304050607 08090A0B0C0D0E0F
v2: 0706050403020100 0F0E0D0C0B0A0908
```

VEC_REVE(ARG1)

Purpose

Returns a vector that contains the elements of the argument in the reverse element order.

Class

Elemental function

Argument type and attributes

ARG1
   An INTENT(IN) integer, unsigned, or real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The result contains the elements of ARG1 in the reverse element order.

Example

```
PROGRAM main

  IMPLICIT NONE

  VECTOR(INTEGER(4)) v1, v2
  INTEGER(4) i4(4)

  i4 = [0, 1, 2, 3]
  v1 = VEC_XL(0, i4)
  CALL VEC_XST(v1, 0, i4)
  PRINT *, 'v1:', i4
  v2 = VEC_REVE(v1)
  CALL VEC_XST(v2, 0, i4)
  PRINT *, 'v2:', i4

END
```

The output:
```
v1: 0 1 2 3
v2: 3 2 1 0
```

VEC_RINT(ARG1)

Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element of the given vector to a floating-point integer.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

Result type and attributes

The result is of the same type as ARG1.

Related reference:

"VEC_ROUNDC(ARG1)" on page 846

VEC_RL(ARG1, ARG2)

Purpose

Rotates each element of a vector left by a given number of bits.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer or unsigned vector.

ARG2
    An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes

The result is of the same type as ARG1.

Result value

Each element of the result is obtained by rotating the corresponding element of ARG1 left by the number of bits specified by the corresponding element of ARG2.

VEC_RLMI(ARG1, ARG2, ARG3)

Purpose

Returns a vector that contains each element of the given vector rotated left and inserted under a mask into another vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) UNSIGNED(4) or UNSIGNED(8) vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

ARG3
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

When VEC_RLMI(ARG1, ARG2, ARG3) is called, each element of ARG1 is rotated left. The rotation count is specified by bits 27-31 of ARG3. After the rotation, ARG1 is inserted under a mask into ARG2. Bits 11-15 of ARG3 contain the mask beginning (mb), and bits 19-23 of ARG3 contain the mask end (me). The mask is generated by setting all bits from bit mb through bit me to 1 and all other bits set to 0.

Note: You cannot specify an all-zero mask.

VEC_RLNM(ARG1, ARG2, ARG3)

Purpose

Returns a vector that contains each element of the given vector rotated left and intersected with a mask.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
 An INTENT(IN) UNSIGNED(4) or UNSIGNED(8) vector

ARG2
 An INTENT(IN) vector of the same type as ARG1

ARG3
 An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

When VEC_RLNM(ARG1, ARG2, ARG3) is called, each element of ARG1 is rotated left and intersected with a mask.

Each element of ARG2 contains the rotation count for the corresponding element of ARG1. The low-order byte of each element is used, and you need to set other bytes to zero.

Each element of ARG3 contains the mask beginning and the mask end for the corresponding element of ARG1. The mask end is in the low-order byte, the mask beginning is in the next higher byte, and you need to set other bytes to zero.

Note: You cannot specify an all-zero mask.

When the data type of the arguments is UNSIGNED(4) vector, no more than 5 bits of each byte should be set for the rotation or the mask beginning and the end; when the data type is UNSIGNED(8) vector, no more than 6 bits of each byte should be set.

VEC_ROUND(ARG1)

Purpose

Returns a vector containing the rounded values of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
  An INTENT(IN) real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the value of the corresponding element of ARG1, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding.

VEC_ROUNDC(ARG1)

Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element in the given vector to integer.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

Result type and attributes

The result is of the same type as `ARG1`.

Related reference:

"VEC_RINT(ARG1)" on page 844

VEC_ROUNDM(ARG1)

Returns a vector containing the largest representable floating-point integer values less than or equal to the values of the corresponding elements of the given vector.

Note: `VEC_ROUNDM` is another name for `VEC_FLOOR`. For details, see "VEC_FLOOR(ARG1)" on page 810.

VEC_ROUNDP(ARG1)

Returns a vector containing the smallest representable floating-point integer values greater than or equal to the values of the corresponding elements of the given vector.

Note: `VEC_ROUNDP` is another name for `VEC_CEIL`. For details, see "VEC_CEIL(ARG1)" on page 786.

VEC_ROUNDZ(ARG1)

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: `VEC_ROUNDZ` is another name for `VEC_TRUNC`. For details, see "VEC_TRUNC(ARG1)" on page 870.

VEC_RSQRT(ARG1)

Purpose

Returns a vector that contains a refined approximation of the reciprocal square roots of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

Result type and attributes

The result is a vector of the same type as `ARG1`.

Result value

Each element of the result contains a refined approximation of the reciprocal square root of the corresponding element of `ARG1`.

Related reference:

"VEC_RSQRTE(ARG1)"

VEC_RSQRTE(ARG1)

Purpose

Returns a vector that contains the estimate value of the reciprocal square roots of the corresponding elements of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An `INTENT(IN)` real vector

Result type and attributes

The result is a vector of the same type as `ARG1`.

Result value

Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of `ARG1`.

VEC_SBOX_BE(ARG1)

Purpose

Performs the SubBytes operation, as defined in *Federal Information Processing Standards FIPS-197*, on a given state `ARG1`.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(1) vector

Result type and attributes

An UNSIGNED(1) vector

Result value

Returns the result of the SubBytes operation.

VEC_SEL(ARG1, ARG2, ARG3)

Purpose

Returns a vector containing the value of either ARG1 or ARG2 depending on the value of ARG3.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer, unsigned, or real vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

ARG3
    An INTENT(IN) unsigned vector containing elements with the same kind as the elements of ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

Each bit of the result vector has the value of the corresponding bit of ARG1 if the corresponding bit of ARG3 is 0, or the value of the corresponding bit of ARG2 otherwise.

VEC_SHASIGMA_BE(ARG1,ARG2,ARG3)

Purpose

Performs a secure hash computation in accordance with *Federal Information Processing Standards FIPS-180-3*, which is a specification for the Secure Hash Standard.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) UNSIGNED(4) or UNSIGNED(8) vector

ARG2

    An INTENT(IN) integer

    The argument must be a constant expression with a value of 0 or 1. It selects the function type, which can be either lowercase sigma (σ) or uppercase sigma (∑).

ARG3

    An INTENT(IN) integer

    The argument must be a constant expression with a value in the range 0 - 15 inclusive. It selects the function subtype, which can be either sigma-0 (σ0 or ∑0) or sigma-1 (σ1 or ∑1).

Result type and attributes

The result has the same type and type parameters as ARG1.

Result value

- If ARG1 is an UNSIGNED(4) vector, for each element i (i = 0,1,2,3) of ARG1, element i of the returned value is the result of the following SHA-256 function:
  - σ0(x[i]), if ARG2 is 0 and bit i of the 4-bit ARG3 is 0
  - σ1(x[i]), if ARG2 is 0 and bit i of the 4-bit ARG3 is 1
  - ∑0(x[i]), if ARG2 is nonzero and bit i of the 4-bit ARG3 is 0
  - ∑1(x[i]), if ARG2 is nonzero and bit i of the 4-bit ARG3 is 1
- If ARG1 is an UNSIGNED(8) vector, for each element i (i = 0,1) of ARG1, element i of the returned value is the result of the following SHA-512 function:
  - σ0(x[i]), if ARG2 is 0 and bit 2*i of the 4-bit ARG3 is 0
  - σ1(x[i]), if ARG2 is 0 and bit 2*i of the 4-bit ARG3 is 1
  - ∑0(x[i]), if ARG2 is nonzero and bit 2*i of the 4-bit ARG3 is 0
  - ∑1(x[i]), if ARG2 is nonzero and bit 2*i of the 4-bit ARG3 is 1

VEC_SL(ARG1, ARG2)

Purpose

Performs a left shift for each element of a vector.

Class

Elemental function

Argument type and attributes

ARG1

    An INTENT(IN) integer or unsigned vector.

ARG2

    An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes

The result is of the same type as ARG1.

Result value

Each element of the result vector is the result of left shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by zeros.

VEC_SLD(ARG1, ARG2, ARG3)

Purpose

Left shifts two concatenated vectors by a given number of bytes.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2
An INTENT(IN) vector of the same type as ARG1

ARG3
An INTENT(IN) integer

It must be specified by a constant expression whose value is in the range 0 - 15 inclusive.

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the most significant 16 bytes obtained by concatenating ARG1 and ARG2, and shifting left by the number of bytes specified by ARG3.

VEC_SLDW(ARG1, ARG2, ARG3)

Purpose

Left shifts two concatenated vectors by a given number of 4 bytes.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) vector

ARG2

An INTENT(IN) vector of the same type as ARG1

ARG3

An INTENT(IN) integer constant expression with the value of 0, 1, 2, or 3

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the most significant 16 bytes obtained by concatenating ARG1 and ARG2, and shifting left by the number of 4 bytes specified by ARG3.

VEC_SLL(ARG1, ARG2)

Purpose

Left shifts a vector by a given number of bits.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) UNSIGNED(1), UNSIGNED(2), or UNSIGNED(4) vector

The least significant three bits of all byte elements in ARG2 must be the same.

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is produced by shifting the contents of ARG1 left by the number of bits specified by the last three bits of the last element of ARG2. The bits that are shifted out are replaced by zeroes.

VEC_SLO(ARG1, ARG2)

Purpose

Left shifts a vector by a given number of bytes.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) unsigned vector whose elements are of kind 1 or 2

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is produced by shifting the contents of ARG1 left by the number of bytes specified by bits 121 through 124 of ARG2. The bits that are shifted out are replaced by zeroes.

VEC_SLV (ARG1, ARG2)

Purpose

Left shifts the elements of a given vector by a given number of bits.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) UNSIGNED(1) vector

ARG2

An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Suppose the following information. Element i of the result vector contains the same bit string as bit S_i - bit S_{i+7} of X_i.
- S_i (0<=i<=15) denotes the value in the three least-significant bits of element i of ARG2.

- X_i (0<=i<=14) denotes the halfword formed by concatenating elements *i* and *i+1* of `ARG1`.
- X_{15} denotes the halfword formed by concatenating element 15 of `ARG1` and a zero byte.

VEC_SPLAT(ARG1, ARG2)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` vector

ARG2
An `INTENT(IN)` integer. Its value must be specified by a constant expression that is greater than or equal to 0, and less than the number of elements in `ARG1`.

Result type and attributes

The result is of the same type as `ARG1`.

Result value

Each element of the result contains the value of the element of `ARG1` that is specified by `ARG2`.

VEC_SPLATS(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` `REAL(4)`, `REAL(8)`, or integer

Result type and attributes

The result is a vector whose elements have the same type as `ARG1`.

Result value

Each element of the result has the value of `ARG1`.

VEC_SPLAT_S8(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an INTEGER(1) vector.

Result value

Each element of the result has the value of ARG1.

VEC_SPLAT_S16(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an INTEGER(2) vector.

Result value

Each element of the result has the value of ARG1.

VEC_SPLAT_S32(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an INTEGER(4) vector.

Result value

Each element of the result has the value of ARG1.

VEC_SPLAT_U8(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an UNSIGNED(1) vector.

Result value

The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the result is given this value.

VEC_SPLAT_U16(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an `UNSIGNED(2)` vector.

Result value

The bit pattern of `ARG1` is interpreted as an unsigned value. Each element of the result is given this value.

VEC_SPLAT_U32(ARG1)

Purpose

Returns a vector of which all the elements are set to a given value.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer. Its value must be specified by a constant expression whose value is in the range -16 to 15 inclusive.

Result type and attributes

The result is an `UNSIGNED(4)` vector.

Result value

The bit pattern of `ARG1` is interpreted as an unsigned value. Each element of the result is given this value.

VEC_SQRT(ARG1)

Purpose

Returns a vector containing the square root of each element in the given vector.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` real vector

Result type and attributes

The result is of the same type as `ARG1`.

VEC_SR(ARG1, ARG2)

Purpose

Performs a right shift for each element of a vector.

Class

Elemental function

Argument type and attributes

ARG1
> An INTENT(IN) integer or unsigned vector.

ARG2
> An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes

The result is of the same type as ARG1.

Result value

Each element of the result vector is the result of right shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by zeros.

VEC_SRA(ARG1, ARG2)

Purpose

Performs an algebraic right shift for each element of a vector.

Class

Elemental function

Argument type and attributes

ARG1
> An INTENT(IN) integer or unsigned vector.

ARG2
> An INTENT(IN) unsigned vector containing elements of the same kind as the elements of ARG1.

Result type and attributes

The result is of the same type as ARG1.

Result value

Each element of the result vector is the result of algebraically right shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits

that are shifted out are replaced by copies of the most significant bit of the element of ARG1.

VEC_SRL(ARG1, ARG2)

Purpose

Right shifts a vector by a given number of bits.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(PIXEL)

ARG2
An INTENT(IN) UNSIGNED(1), UNSIGNED(2), or UNSIGNED(4) vector

Note: The least significant three bits of all byte elements in ARG2 must be the same.

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is produced by shifting the contents of ARG1 right by the number of bits specified by the last three bits of the last element of ARG2. The bits that are shifted out are replaced by zeroes.

VEC_SRO(ARG1, ARG2)

Purpose

Right shifts a vector by a given number of bytes.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2
　　An INTENT(IN) unsigned vector whose elements are of kind 1 or 2

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is produced by shifting the contents of ARG1 right by the number of bytes specified by bits 121 through 124 of ARG2. The bits that are shifted out are replaced by zeroes.

VEC_SRV (ARG1, ARG2)

Purpose

Right-shifts the elements of a given vector by a given number of bits.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
　　An INTENT(IN) UNSIGNED(1) vector

ARG2
　　An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Suppose the following information. Element i of the result vector contains the same bit string as from bit 8-S_i to bit 15-S_i of X_i, inclusive.

- S_i (0<=i<=15) denotes the value in the three least-significant bits of element i of ARG2.
- X_0 denotes the value of the halfword formed by concatenating a zero byte and element 0 of ARG1.
- X_i (1<=i<=15) denotes the value of the halfword formed by concatenating elements i and $i+1$ of ARG1.

VEC_ST(ARG1, ARG2, ARG3)

Purpose

Stores a vector to memory at the given address.

Class

Pure subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) integer

ARG3

is INTENT(OUT). It must be a vector or be of type integer or real as follows:

Case (i)

If ARG3 is a vector, it must be of the same type as ARG1.

Case (ii)

If ARG3 is not a vector, and ARG1 is an integer vector or an unsigned vector, then ARG3 must be of type integer with the same kind type parameter as the elements of ARG1.

Case (iii)

If ARG3 is not a vector, and ARG1 is a real vector, then ARG3 must be of the same type and kind as the elements of ARG1.

Case (iv)

If ARG3 is not a vector, and ARG1 is a pixel vector, then ARG3 must be of type INTEGER(2).

Result value

The VEC_ST function returns nothing. ARG2 is added to the address of ARG3, and the sum is truncated to a multiple of 16 bytes. The value of ARG1 is then stored into this memory address.

VEC_STE(ARG1, ARG2, ARG3)

Purpose

Stores a vector element into memory at the given address.

Class

Subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))

- VECTOR(PIXEL)

ARG2

An INTENT(IN) integer vector

ARG3

Is INTENT(OUT).

- If ARG1 is a pixel vector, ARG3 must be of type INTEGER(2).
- If ARG1 is an unsigned vector, ARG3 must be of type integer and must have the same kind as the elements of ARG1.
- Otherwise, ARG3 must have the same type and kind as the elements of ARG1.

Result value

The effective address is the sum of ARG2 and the address specified by ARG3, truncated to a multiple of the size in bytes of an element of the result vector. The value of the element of ARG1 at the byte offset that corresponds to the four least significant bits of the effective address is stored into memory at the effective address.

Related information:

[pdf] -qaltivec

[pdf] Vector element order toggling

VEC_STL(ARG1,ARG2,ARG3)

Purpose

Stores a vector into memory at the given address, and marks the data as Least Recently Used.

Class

Subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:

- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), or VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

An INTENT(IN) integer vector

ARG3

Is INTENT(OUT). It must be a vector or be of type integer or real as follows:

Case (i)

If ARG3 is a vector, it must be of the same type as ARG1.

Case (ii)

If ARG3 is not a vector, and if ARG1 is an integer vector or an unsigned vector, ARG3 must be of type integer with the same kind type parameter as the elements of ARG1.

Case (iii)

> If ARG3 is not a vector, and if ARG1 is a real vector, ARG3 must be of the same type and kind as the elements of ARG1.

Case (iv)

> If ARG3 is not a vector, and if ARG1 is a pixel vector, ARG3 must be of type `INTEGER(2)`.

Result value

ARG2 is added to the address specified by ARG3, and the sum is truncated to a multiple of 16 bytes. The value of ARG1 is then stored into this memory address. The data is marked as Least Recently Used.

Related information:

📄 -qaltivec

📄 Vector element order toggling

VEC_SUB(ARG1, ARG2)

Purpose

Returns a vector containing the result of subtracting each element of ARG2 from the corresponding element of ARG1.

Note: For `INTEGER(8)` and `UNSIGNED(8)` vectors, this function emulates the operation.

Class

Elemental function

Argument type and attributes

ARG1
An `INTENT(IN)` integer, unsigned, or real vector

ARG2
An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The arithmetic is modular for integer vectors.

VEC_SUB_U128 (ARG1, ARG2)

Purpose

Subtracts unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) VECTOR(UNSIGNED(1))

ARG2
An INTENT(IN) VECTOR(UNSIGNED(1))

Result type and attributes

Same as *ARG1*.

Result value

Let a and b be the 128-bit integers that are storage-associated with *ARG1* and *ARG2*, respectively.

The result value is a - b.

VEC_SUBC_U128 (ARG1, ARG2)

Purpose

Returns the carry bit of the 128-bit subtraction of two quadword values.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) VECTOR(UNSIGNED(1))

ARG2
An INTENT(IN) VECTOR(UNSIGNED(1))

Result type and attributes

Same as *ARG1*.

Result value

Let a and b be the 128-bit integers that are storage-associated with *ARG1* and *ARG2*, respectively.

The result value is the carry out of a - b.

VEC_SUBE_U128 (ARG1, ARG2, ARG3)

Purpose

Subtracts unsigned quadword values with carry bit from previous operation.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) VECTOR(UNSIGNED(1))

ARG2
    An INTENT(IN) VECTOR(UNSIGNED(1))

ARG3
    An INTENT(IN) VECTOR(UNSIGNED(1))

Result type and attributes

Same as *ARG1*.

Result value

Let a, b, c be the 128-bit integers that are storage-associated with *ARG1*, *ARG2*, and *ARG3*, respectively.

The result value is a - b - iand(c, 1) .

VEC_SUBEC_U128 (ARG1, ARG2, ARG3)

Purpose

Gets the carry bit of the 128-bit subtraction of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) VECTOR(UNSIGNED(1))

ARG2
   An `INTENT(IN) VECTOR(UNSIGNED(1))`

ARG3
   An `INTENT(IN) VECTOR(UNSIGNED(1))`

Result type and attributes

Same as *ARG1*.

Result value

Let a, b, c be the 128-bit integers that are storage-associated with *ARG1*, *ARG2*, and *ARG3*, respectively.

The result value is the carry out of `a - b - iand(c, 1)`.

VEC_SUBC(ARG1, ARG2)

Purpose

Returns a vector containing the borrows produced by subtracting each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
   An `INTENT(IN) UNSIGNED(4)` vector

ARG2
   An `INTENT(IN)` vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the value of the borrow produced by subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The value is 0 if a borrow occurred, or 1 if no borrow occurred.

VEC_SUBS(ARG1, ARG2)

Purpose

Returns a vector containing the saturated differences of each set of corresponding elements of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) integer or unsigned vector whose elements are of kind 1, 2, or 4

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The value of each element of the result is the saturated result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1.

VEC_SUM2S(ARG1, ARG2)

Purpose

Returns a vector containing the results of performing a sum across 1/2 vector operation on two given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) INTEGER(4) vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The first and third elements of the result are 0. The second element of the result contains the saturated sum of the first and second elements of ARG1 and the second element of ARG2. The fourth element of the result contains the saturated sum of the third and fourth elements of ARG1 and the fourth element of ARG2.

Related information:

 -qaltivec

 Vector element order toggling

VEC_SUM4S(ARG1, ARG2)

Purpose

Returns a vector containing the results of performing a sum across 1/4 vector operation on two given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) INTEGER(1), INTEGER(2), or UNSIGNED(1) vector

ARG2
    An INTENT(IN) vector

    If ARG1 is an integer vector, ARG2 is an INTEGER(4) vector. If ARG1 is an unsigned vector, ARG2 is an UNSIGNED(4) vector.

Result type and attributes

The following table describes the types of the result.

Table 84. Types of the returned value and function arguments

data type of result	data type of ARG1	data type of ARG2
An INTENT(IN) INTEGER(4) vector	An INTENT(IN) INTEGER(1) vector	An INTENT(IN) INTEGER(4) vector
An INTENT(IN) UNSIGNED(4) vector	An INTENT(IN) UNSIGNED(1) vector	An INTENT(IN) UNSIGNED(4) vector
An INTENT(IN) INTEGER(4) vector	An INTENT(IN) INTEGER(2) vector	An INTENT(IN) INTEGER(4) vector

Result value

For each element n of the result vector, the value is obtained as follows:
- If ARG1 is an INTEGER(1) or UNSIGNED(1) vector, the value is the saturated addition of elements 4n through 4n+3 of ARG1 and element n of ARG2.
- If ARG1 is an INTEGER(2) vector, the value is the saturated addition of elements 2n through 2n+1 of ARG1 and element n of ARG2.

VEC_SUMS(ARG1, ARG2)

Purpose

Returns a vector containing the results of performing a sum across vector operation on the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) INTEGER(4) vector

ARG2
    An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

The first three elements of the result are 0. The fourth element is the saturated sum of all the elements of ARG1 and the fourth element of ARG2.

Related information:

 -qaltivec

 Vector element order toggling

VEC_TEST_DATA_CLASS(ARG1, ARG2)

Purpose

Determines the data class of the elements of the given vector.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

ARG2
    An INTENT(IN) const int

Result type and attributes

The following table describes the type of the result.

Table 85. Types of the returned value and function arguments

result	ARG1	ARG2
UNSIGNED(4) vector	REAL(4) vector	const int
UNSIGNED(8) vector	REAL(8) vector	const int

Result value

Returns the results of testing ARG1 for the condition selected by ARG2. The value of ARG2 is in the range of 0-127. Each bit of ARG2 enables the test of a different condition. You can refer to the following table for the mapping relations between testing conditions and bits of ARG2:

Table 86. Bitmask for testing conditions

Bits of ARG2	Test conditions
0x01	Test for -Denormal
0x02	Test for +Denormal
0x04	Test for -Zero
0x08	Test for +Zero
0x10	Test for -Infinity
0x20	Test for +Infinity
0x40	Test for NaN

If any of the enabled test conditions is true, all of the bits of corresponding element are set to 1. If all of the enabled test conditions are false, all of the bits of the corresponding element is set to 0.

VEC_TRUNC(ARG1)

Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: VEC_TRUNC is another name for VEC_ROUNDZ. For details, see "VEC_ROUNDZ(ARG1)" on page 847.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) real vector

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result contains the value of the corresponding element of ARG1, truncated to an integral value.

VEC_UNPACKH(ARG1)

Purpose

Unpacks the most significant half of a vector into another vector.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4)[1], or PIXEL vector.

    Note: When you call VEC_UNPACKH, vectors of INTEGER(4) are valid only when -qarch is set to target POWER8 or higher processors.

Result type and attributes

The following table describes the types of the result.

Table 87. Types of the returned value and function arguments

d	a	
INTEGER(2) vector	An INTENT(IN) INTEGER(1) vector	
INTEGER(4) vector	An INTENT(IN) INTEGER(2) vector	
INTEGER(8) vector	An INTENT(IN) INTEGER(4) vector[2]	
UNSIGNED(4) vector	An INTENT(IN) pixel vector	

Note: When you call VEC_UNPACKH, vectors of INTEGER(4) are valid only when -qarch is set to target POWER8 or POWER9 processors.

Result value

If ARG1 is an integer vector, the value of each element of the result is the value of the corresponding element of the most significant half of ARG1.

If ARG1 is a pixel vector, the value of each element of the result is taken from the corresponding element of the most significant half of ARG1 as follows: all bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1; the least significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.

VEC_UNPACKL(ARG1)

Purpose

Unpacks the least significant half of a vector into another vector.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) INTEGER(1), INTEGER(2), INTEGER(4)[1], or PIXEL vector.

    Note: When you call VEC_UNPACKL, vectors of INTEGER(4) are valid only when -qarch is set to target POWER8 or higher processors.

Result type and attributes

The following table describes the types of the result and the function arguments.

Table 88. Types of the returned value and function arguments

d	a	
INTEGER(2) vector	An INTENT(IN) INTEGER(1) vector	
INTEGER(4) vector	An INTENT(IN) INTEGER(2) vector	
INTEGER(8) vector	An INTENT(IN) INTEGER(4) vector[2]	
UNSIGNED(4) vector	An INTENT(IN) pixel vector	

Note: When you call VEC_UNPACKL, vectors of INTEGER(4) are valid only when -qarch is set to target POWER8 or higher processors.

Result value

If ARG1 is an integer vector, then the value of each element of the result is the value of the corresponding element of the least significant half of ARG1. If ARG1 is a pixel vector, then the value of each element of the result is taken from the corresponding element of the least significant half of ARG1 as follows: all bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1; the least significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1; the least significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.

VEC_VCLZ(ARG1)

Purpose

Computes the count of leading zero bits of each element of the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) integer or unsigned vector.

Result type and attributes

The result is a vector of the same type as ARG1.

Result value

Each element of the result is set to the number of leading zeros of the corresponding element of ARG1.

Related reference:

"VEC_CNTLZ(ARG1)" on page 794

VEC_VGBBD(ARG1)

Purpose

Performs a gather-bits-by-bytes operation on the given vector.

Class

Elemental function

Argument type and attributes

ARG1
    An INTENT(IN) VECTOR(UNSIGNED(1)) or VECTOR(SIGNED(1)).

Result type and attributes

The result is of the same type as ARG1.

Result value

Each doubleword element of the result is set as follows:

Let x(i) (0 <= i < 8) denote the byte elements of the corresponding input doubleword element, with x(7) as the most significant byte. For each pair of i and j (0 <= i < 8, 0 <= j < 8), the jth bit of the ith byte element of the result is set to the value of the ith bit of the jth byte element of the input.

Related reference:

"VEC_GBB(ARG1)" on page 811

VEC_XL(ARG1, ARG2)

Purpose

Loads a 16-byte vector from the memory address specified by the displacement ARG1 and the pointer ARG2.

Class

Pure function

Argument type and attributes

ARG1

   An INTENT(IN) integer

ARG2

   An INTENT(IN) variable of any of the following types:
   - REAL(4), REAL(8)
   - INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)
   - Vector

Result type and attributes

If ARG2 is a vector, the result is of the same type as ARG2; otherwise, the result is a vector of the ARG2 type.

Result value

VEC_XL adds the displacement provided by ARG1 to the address provided by ARG2 to obtain the effective address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

This function loads the result from a specific number of elements at the effective address for the load operation. The number and the size of elements vary based on the data type of argument ARG2. For details about the function result value, see the following table.

Table 89. Function result value

Data type of ARG2	Function result value
INTEGER(1)	Loads the result from 16 one-byte elements.
INTEGER(2)	Loads the result from 8 two-byte elements.
INTEGER(4) or REAL(4)	Loads the result from four 4-byte elements.
INTEGER(8) or REAL(8)	Loads the result from two 8-byte elements.
VECTOR	Loads the result from corresponding elements.

The order of elements in the function result is big endian when -qaltivec=be is in effect. Otherwise, the order is little endian.

VEC_XL_BE(ARG1, ARG2)

Purpose

Loads a 16-byte vector from the memory address specified by the displacement ARG1 and the pointer ARG2.

Class

Pure function

Argument type and attributes

ARG1

   An INTENT(IN) integer

ARG2
> An INTENT(IN) variable of any of the following types:
> - REAL(4), REAL(8)
> - INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)
> - Vector

Result type and attributes

If ARG2 is a vector, the result is of the same type as ARG2; otherwise, the result is a vector of ARG2 type.

Result value

VEC_XL_BE adds the displacement provided by ARG1 to the address provided by ARG2 to obtain the effective address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

This function loads the result from a specific number of elements at the effective address for the load operation function. The number and the size of elements vary based on the data type of argument ARG2. For details about the function result value, see the following table.

Table 90. Function result value

Data type of ARG2	Function result value
INTEGER(1)	Loads the result from 16 one-byte elements.
INTEGER(2)	Loads the result from 8 two-byte elements.
INTEGER(4) or REAL(4)	Loads the result from four 4-byte elements.
INTEGER(8) or REAL(8)	Loads the result from two 8-byte elements.
VECTOR	Loads the result from corresponding elements.

The order of elements in the function result is big endian regardless of the -qaltivec option in effect.

Example: VEC_XL_BE function loads the vector elements in register

The following code example shows that the **VEC_XL_BE** function loads the vector elements in registers always in big endian element order. The **VEC_EXTRACT** function extracts the vector elements in the proper element order specified by -qaltivec=be or -qaltivec=le.

Assume that the file vec_xl_be.f contains the following codes.

```
PROGRAM main

 IMPLICIT NONE

 VECTOR(INTEGER(4)) a4
 INTEGER(4) c4(4)
 INTEGER i

 c4 = [0, 1, 2, 3]
 a4 = VEC_XL_BE(0, c4)
 PRINT *, (VEC_EXTRACT(a4, i), i=0, 3)

END
```

- Compile the codes with option -qaltivec=le by running the following command:

```
xlf95 vec_xl_be.f -qaltivec=le
```

Output:

```
3 2 1 0
```

- Compile the program with option -qaltivec=be by running the following command:

```
xlf95 vec_xl_be.f -qaltivec=be
```

Output:

```
0 1 2 3
```

VEC_XL_LEN(ARG1, ARG2)

Purpose

Returns a vector that loads a given number of bytes from the given address.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument type and attributes

ARG1

An INTENT(IN) variable of any of the following types:

- INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)
- REAL(4), or REAL(8)
- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4)), or VECTOR(INTEGER(8))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4)), or VECTOR(UNSIGNED(8))
- VECTOR(REAL(4)), or VECTOR(REAL(8))

ARG2

An INTENT(IN) integer of kind 1, 2, 4 or 8

Result type and attributes

If ARG1 is a vector, the result is of the same type as ARG1; otherwise the result is a vector with elements of the same type as ARG1.

Result value

The result is loaded from the memory address that is specified by ARG1. The number of bytes loaded is specified by ARG2. Bytes of elements are initialized in order from the byte stream which is defined by the endianness of the operating environment. Any byte of elements that is not initialized is set to a 0 value.

The behavior is undefined if the value of ARG2 is outside of the range 0 - 255.

VEC_XLD2(ARG1, ARG2)

Purpose

Loads a 16-byte vector from two 8-byte elements at the memory address specified by the displacement `ARG1` and the pointer `ARG2`.

This function is deprecated and has been replaced with `VEC_XL(ARG1, ARG2)`.

Class

Pure function

Argument type and attributes

ARG1
    An `INTENT(IN)` integer

ARG2
    An `INTENT(IN)` variable. The variable can be any of the following types:
- `REAL(4)` or `REAL(8)`
- `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, or `INTEGER(8)`
- Vector

Result type and attributes

If `ARG2` is a vector, the result is of the same type as `ARG2`; otherwise the result is a vector of the `ARG2` type.

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:
VEC_XL(ARG1, ARG2)

Related information:

 -qaltivec

 Vector element order toggling

VEC_XLDS(ARG1, ARG2)

Purpose

Loads an 8-byte element from the memory address specified by the displacement `ARG1` and the pointer `ARG2` and then splats it onto a vector.

Class

Pure function

Argument type and attributes

ARG1
　　An `INTENT(IN)` integer

ARG2
　　An `INTENT(IN)` variable. The variable can be any of the following types:
- `REAL(8)` or `INTEGER(8)`
- `VECTOR(INTEGER(8))`, `VECTOR(UNSIGNED(8))`, or `VECTOR(REAL(8))`

Result type and attributes

The following table describes the result types of the function.

Table 91. Result types for VEC_XLDS

ARG1 type	ARG2 type	Result type
integer	`VECTOR(REAL(8))`	`VECTOR(REAL(8))`
	`REAL(8)`	
	`REAL(8)`	`VECTOR(INTEGER(8))`
	`INTEGER(8)`	
	`VECTOR(INTEGER(8))`	
	`VECTOR(UNSIGNED(8))`	`VECTOR(UNSIGNED(8))`

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

VEC_XLW4(ARG1, ARG2)

Purpose

Loads a 16-byte vector from four 4-byte elements at the memory address specified by the displacement ARG1 and the pointer ARG2.

Notes:
- It only differs in semantics from `VEC_XLD2` on a little endian machine.
- This function is deprecated and has been replaced with `VEC_XL(ARG1, ARG2)`.

Class

Pure function

Argument type and attributes

ARG1
　　An `INTENT(IN)` integer

ARG2
　　An `INTENT(IN)` variable. The variable can be any of the following types:
- `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, or `REAL(4)`
- `VECTOR(INTEGER(1))` or `VECTOR(INTEGER(2))`

- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), or VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

Result type and attributes

If ARG2 is a vector, the result is of the same type as ARG2; otherwise the result is a vector of the ARG2 type.

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:
VEC_XL(ARG1, ARG2)

Related information:

📄 -qaltivec

📄 Vector element order toggling

VEC_XOR(ARG1, ARG2)

Purpose

Performs a bitwise XOR of the given vectors.

Class

Elemental function

Argument type and attributes

ARG1
An INTENT(IN) vector

ARG2
An INTENT(IN) vector of the same type as ARG1

Result type and attributes

The result is of the same type as ARG1.

Result value

The result is the bitwise XOR of ARG1 and ARG2.

VEC_XST(ARG1, ARG2, ARG3)

Purpose

Stores the elements of the 16-byte vector ARG1 to the effective address obtained by adding the displacement provided by ARG2 with the address provided by ARG3. The

effective address is not truncated to a multiple of 16 bytes.

Class

Pure subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector.

The -qaltivec option affects the element order of argument ARG1.

ARG2

An INTENT(IN) integer

ARG3

An INTENT(OUT) variable. The variable must be a vector or of type integer or real as follows:

- If ARG3 is a vector, ARG3 must be of the same type as ARG1.
- If ARG3 is not a vector, and ARG1 is an integer vector or an unsigned vector, ARG3 must be of integer type with the same kind type parameter as the elements of ARG1.
- If ARG3 is not a vector and ARG1 is a real vector, ARG3 must be of the same type and kind as the elements of ARG1.
- If ARG3 is not a vector and ARG1 is a pixel vector, ARG3 must be of type INTEGER(2).

VEC_XST_BE(ARG1, ARG2, ARG3)

Purpose

Stores the elements of the 16-byte vector ARG1 in big endian element order to the effective address obtained by adding the displacement provided by ARG2 with the address provided by ARG3. The effective address is not truncated to a multiple of 16 bytes.

Class

Pure subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector

The element sequence of vector ARG1 is always in big endian order regardless of the -qaltivec option value.

ARG2

An INTENT(IN) integer

ARG3

An INTENT(OUT) variable

The variable must be a vector or of type integer or real as follows:

- If ARG3 is a vector, ARG3 must be of the same type as ARG1.

- If ARG3 is not a vector, and ARG1 is an integer vector or an unsigned vector, ARG2 must be of integer type with the same kind type parameter as the elements of ARG1.
- If ARG3 is not a vector and ARG1 is a real vector, ARG3 must be of the same type and kind as the elements of ARG1.
- If ARG3 is not a vector and ARG1 is a pixel vector, ARG3 must be of type INTEGER(2).

VEC_XST_LEN(ARG1, ARG2, ARG3)

Purpose

Stores a given byte length of a vector to a given address.

This intrinsic procedure is valid only when -qarch is set to utilize POWER9 technology.

Class

Elemental function

Argument types

The following table describes the types of the function arguments.

Table 92. Types of function arguments

ARG1	ARG2	ARG3
INTEGER(1) vector	INTEGER(1), POINTER	SIZE_T
UNSIGNED(1) vector	UNSIGNED(1), POINTER	SIZE_T
INTEGER(2) vector	INTEGER(2), POINTER	SIZE_T
UNSIGNED(2) vector	UNSIGNED(2), POINTER	SIZE_T
INTEGER(4) vector	INTEGER(4), POINTER	SIZE_T
UNSIGNED(4) vector	UNSIGNED(4), POINTER	SIZE_T
INTEGER(8) vector	INTEGER(8), POINTER	SIZE_T
UNSIGNED(8) vector	UNSIGNED(8), POINTER	SIZE_T
REAL(4) vector	REAL(4), POINTER	SIZE_T
REAL(8) vector	REAL(8), POINTER	SIZE_T

Result

No value is returned. The specified bytes of ARG1 are stored into an address. The number of bytes to be stored is specified by ARG3 and the address is specified by ARG2.

The behavior is undefined if the value of ARG3 is outside of 0-255. The behavior is implementation defined if the value of ARG3 is not a multiple of the vector element size.

VEC_XSTD2(ARG1, ARG2, ARG3)

Purpose

Puts a 16-byte vector ARG3 as two 8-byte elements to the memory address specified by the displacement ARG2 and the pointer ARG1.

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the effective address to a multiple of 16 bytes.

This function is deprecated and has been replaced with VEC_XST(ARG1, ARG2, ARG3).

Class

Pure subroutine

Argument type and attributes

ARG1
An INTENT(IN) vector

ARG2
An INTENT(IN) integer

ARG3
An INTENT(OUT) vector of the same type as ARG1

Related reference:
VEC_XST(ARG1, ARG2, ARG3)

Related information:

 -qaltivec

 Vector element order toggling

VEC_XSTW4(ARG1, ARG2, ARG3)

Purpose

Puts a 16-byte vector ARG3 as four 4-byte elements to the memory address specified by the displacement ARG1 and the pointer ARG2.

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the effective address to a multiple of 16 bytes.

This function is deprecated and has been replaced with VEC_XST(ARG1, ARG2, ARG3).

Class

Pure subroutine

Argument type and attributes

ARG1

An INTENT(IN) vector of any of the following types:

- VECTOR(INTEGER(1)), VECTOR(INTEGER(2)), VECTOR(INTEGER(4))
- VECTOR(UNSIGNED(1)), VECTOR(UNSIGNED(2)), VECTOR(UNSIGNED(4))
- VECTOR(REAL(4))
- VECTOR(PIXEL)

ARG2

INTENT(IN) integer

ARG3

INTENT(OUT) vector of the same type as ARG1

Related reference:

VEC_XST(ARG1, ARG2, ARG3)

Related information:

 -qaltivec

 Vector element order toggling

Chapter 18. Language interoperability features

XL Fortran provides a standardized mechanism for interoperating with C based on the Fortran 2003 Standard. An entity is said to be interoperable if equivalent declarations of it can be made in the two languages. XL Fortran enforces interoperability for types, variables, common blocks, and procedures. Interoperability with the C programming language allows portable access to many libraries and the low-level facilities provided by C and allows the portable use of Fortran libraries by programs written in C. Details of this implementation are discussed in this section.

Interoperability of types

Intrinsic types

XL Fortran provides the **ISO_C_BINDING** intrinsic module that contains named constants holding kind type parameter values for intrinsic types. Their names are shown together with the corresponding C types in Table 93 on page 900. Only those intrinsic types listed in the table are interoperable; other intrinsic types are not.

Derived types

XL Fortran provides the ability to define derived types that correspond to C **struct** types. A Fortran derived type with the **BIND** attribute is interoperable with a C **struct** type if all of the following conditions are met:

- The Fortran derived type definition is given the **BIND(C)** attribute explicitly.
- The Fortran derived type and C **struct** type have the same number of components.
- The components of the Fortran derived type have types and type parameters that are interoperable with the types of the corresponding components of the C **struct** type, and cannot have the **POINTER** or **ALLOCATABLE** attribute.
- The components of the Fortran derived type and of the C **struct** type are declared in the same relative positions in their relative type definitions.

For example, the C type *myctype*, declared below, is interoperable with the Fortran type *myftype*, declared below.

```
typedef struct {
  int m, n;
  float r;
} myctype;
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
  INTEGER(C_INT) :: I, J
  REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

Note that the names of the corresponding components of the derived type and the C struct type need not be the same; the names are not significant in determining whether the Fortran derived type and C **struct** type are interoperable.

There is no Fortran type that is interoperable with a C struct type that contains a bit field or that contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

Interoperability of variables

A Fortran module variable that has the **BIND** attribute may interoperate with a C variable with external linkage.

Fortran module variables

There need not be an associated C entity for a module variable with the **BIND** attribute.

Scalar variables

A scalar Fortran variable is interoperable if its type and type parameters are interoperable and it has neither the **POINTER** nor the **ALLOCATABLE** attributes. An interoperable scalar Fortran variable is interoperable with a scalar C variable if its type and type parameters are interoperable with the type of the C variable.

Array variables

A Fortran array variable is interoperable, if its type and type parameters are interoperable, it is of explicit shape or assumed size, it is not zero-sized, and it does not have the **POINTER** or **ALLOCATABLE** attributes.

A Fortran array is interoperable with a C array, if its size is nonzero and
• Its rank is equal to one and an element of the array is interoperable with an element of the C array
• Its rank is greater than one and the base types of the two arrays are equivalent and each of the dimensions correspond.

Because C uses row-major arrays and Fortran uses column-major arrays, a C array's dimensions must be the reverse of a Fortran array's dimensions.

Interoperable variables in asynchronous communication (TS 29113)

Asynchronous communication for a Fortran variable can occur when procedures that are defined by means other than Fortran are called. You must specify the **ASYNCHRONOUS** attribute for the variables that are used for the asynchronous communication. For example, you must mark a variable with the **ASYNCHRONOUS** attribute if the variable is used in a nonblocking Message Passing Interface (MPI) call.

For the variables in asynchronous communication, the asynchrony is initiated by the execution of an asynchronous communication initiation procedure, and the asynchrony is completed by the execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, a pending communication affector exists. The variable that participates in the asynchronous communication is a pending communication affector. Any variable whose part is associated with the pending communication affector or with a part of the pending communication affector is also a pending communication affector.

Asynchronous communication is either input communication or output communication.

Restrictions:

- For input communication, a pending communication affector cannot be referenced, become defined, become undefined, become associated with a dummy argument that has the **VALUE** attribute, or have its pointer association status changed.
- For output communication, a pending communication affector cannot be redefined, become undefined, or have its pointer association status changed.

Note: To avoid unnecessary reduction in performance and to limit the scope in which a variable has the **ASYNCHRONOUS** attribute, you can use the **BLOCK** construct to surround the asynchronous communication initiation and completion procedures and specify the **ASYNCHRONOUS** attribute for the communication affector within the **BLOCK** construct. See the following example for details.

```
PROGRAM sum2arrays
  IMPLICIT NONE
  INCLUDE 'mpif.h'

  INTEGER :: mpierr, i
  INTEGER :: status(MPI_STATUS_SIZE)
  INTEGER, PARAMETER :: DIM = 100
  INTEGER, PARAMETER :: TAG_SEND_ARR = 10, TAG_RES_READY = 11
  REAL :: arraya(DIM)
  REAL :: arrayb(DIM)
  INTEGER :: reqs(2)
  REAL :: partialSum1 = 0, partialSum2=0

  ! initialization
  CALL MPI_INIT (mpierr)

  ! Assume there are two tasks in the world communicator:
  ! Task 0: to compute the sum of arrayb and the sum of square of arrayb
  ! Task 1: to compute the sum of arraya
  IF (rank .eq. 0) THEN
    arraya = [(real(i), i=1, DIM, 1)]
    BLOCK
      ASYNCHRONOUS :: arraya
      CALL MPI_ISEND(arraya, DIM, MPI_REAL, 1, TAG_SEND_ARR, &
                     MPI_COMM_WORLD, reqs(1), mpierr)

      ! While waiting for the arraya to be sent, you
      ! can do some calculations for arrayb.
      arrayb = [(real(i+1*2), i=1, DIM, 1)]
      partialSum1 = SUM(arrayb)

      CALL MPI_WAIT(reqs(1), status, mpierr)

      ! Now arraya has been sent. You can modify it.
      arraya = arrayb**2
      PRINT *, "Sum(b**2) = ", SUM(arraya)
    END BLOCK

    ! Get the sum calculated by task 1.
    CALL MPI_RECV(partialSum2, 1, MPI_REAL, 1, TAG_RES_READY, &
                  MPI_COMM_WORLD, status, mpierr)

    ! The final result is computed by adding the two
    ! values that are computed by task 0 and task 1.
    PRINT *, "Sum(a) + Sum(b) = ", partialSum1 + partialSum2
```

```
      END IF

      CALL MPI_FINALIZE(mpierr)
    END
```

Related information
• "ASYNCHRONOUS" on page 293

Interoperability of common blocks

A C variable with external linkage can interoperate with a common block that has the **BIND** attribute.

If a common block has the **BIND** attribute, it must have the **BIND** attribute and the same binding label in each scoping unit in which it is declared. A C variable with external linkage interoperates with a common block with the **BIND** attribute if:
• The C variable is of a struct type and the variables that are members of the common block are interoperable with corresponding components of the struct type, or
• The common block contains a single variable, and the variable is interoperable with the C variable.

There need not be an associated C entity for a common block with the **BIND** attribute.

Interoperability of procedures

A Fortran procedure is interoperable if it has the **BIND** attribute. A Fortran procedure interface is interoperable with a C function prototype if the following conditions are met:
• The interface has the **BIND(C)** attribute.
• The interface describes a function whose result variable is a scalar variable that is interoperable with the result of the prototype, or the interface describes a subroutine, and the prototype has a result type of `void`.
• The number of dummy arguments of the interface is equal to the number of formal parameters of the prototype.
• Any scalar dummy argument with the **VALUE** attribute is interoperable with the corresponding formal parameter of the prototype.
• Any dummy argument without the **VALUE** attribute corresponds to a formal parameter of the prototype that is of a pointer type, and satisfies one of the following conditions:
  – The dummy argument is interoperable with an entity of the referenced type of the formal parameter.
  – **TS 29113** The dummy argument is assumed-type and not allocatable or pointer. It is also not assumed-shape or assumed-rank. It corresponds to a formal parameter that is a pointer to `void`.
  – The dummy argument is allocatable or a pointer data object without the **CONTIGUOUS** attribute. It is interoperable with a formal parameter of the prototype that is a pointer to type `CFI_cdesc_t`.
  – The dummy argument is an assumed-rank object that is interoperable with the actual argument of a C function.
  – The dummy argument is optional.

– The dummy argument is a nonallocatable and nonpointer object of character type with assumed-length. It corresponds to a formal parameter of the prototype that is a pointer to type `CFI_cdesc_t`. TS 29113

• The prototype does not have variable arguments.

Example

In the following example, the Fortran procedure interface:

```
INTERFACE
   FUNCTION func(i, j, k, l, m) BIND(C)
     USE, INTRINSIC :: ISO_C_BINDING
     INTEGER(C_SHORT) :: func
     INTEGER(C_INT), VALUE :: i
     REAL(C_DOUBLE) :: j
     INTEGER(C_INT) :: k, l(10)
     TYPE(C_PTR), VALUE :: m
   END FUNCTION func

END INTERFACE
```

is interoperable with the C function prototype:

```
short func(int i, double *j, int *k, int l[10], void *m);
```

A C data pointer might correspond to a Fortran dummy argument of type **C_ PTR** or to a Fortran scalar that does not have the **VALUE** attribute. In the example, the C pointers *j* and *k* correspond to the Fortran scalars **J** and **K**. The C pointer **m** corresponds to the Fortran dummy argument **M** of type **C_ PTR**.

Related information:

Type CFI_cdesc_t

Interoperability of optional arguments (TS 29113)

Procedures that have optional dummy arguments can interoperate with C functions. You can specify the **OPTIONAL** attribute for a dummy argument in a procedure interface that has the **BIND(C)** attribute.

With this feature, you can call Fortran procedures with optional dummy arguments from C. You can also call a C function, with a corresponding interoperable Fortran interface that has one or more optional dummy arguments, from Fortran.

For example, the following subroutine interface has one optional dummy argument `arg`:

```
INTERFACE
  SUBROUTINE sub(arg) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_INT), OPTIONAL :: arg
  END SUBROUTINE
END INTERFACE
```

Restriction: An interoperable dummy argument cannot have both the **OPTIONAL** and **VALUE** attributes.

Interoperability of allocatable and pointer arguments (TS 29113)

Procedures that have allocatable or pointer dummy arguments can interoperate with C functions. You can specify the **ALLOCATABLE** or **POINTER** attributes for a dummy argument in a procedure interface that has the **BIND(C)** attribute.

With this feature, you can perform the following tasks:
- Call Fortran procedures with allocatable and pointer dummy arguments from C.
- Pass Fortran objects with **ALLOCATABLE** or **POINTER** attributes to a C function.
- Construct C descriptors to represent allocatable and pointer objects and pass them back to Fortran.

Restrictions:
- You cannot specify the **ALLOCATABLE** or **POINTER** attributes for a default-initialized dummy argument of a procedure that has the **BIND(C)** attribute.
- If the allocatable or pointer dummy argument in an interoperable interface is of type character, its character length must be deferred.
- A pointer dummy argument in an interoperable interface cannot have the **CONTIGUOUS** attribute.

Related information
- "ALLOCATABLE (Fortran 2003)" on page 285
- "BIND (Fortran 2003)" on page 298
- "Character" on page 44
- "CONTIGUOUS (Fortran 2008)" on page 323
- "POINTER (Fortran 90)" on page 426

Rules for allocatable and pointer arguments

The following rules apply when arguments for an interoperable procedure have **ALLOCATABLE** or **POINTER** attributes.

Change of allocation or association status

The following rules apply to the change of allocation and association status:
- Within a C function, you can allocate or deallocate an allocatable object only by calling the `CFI_allocate` and `CFI_deallocate` functions.
- You can associate a Fortran pointer with a target by calling the `CFI_allocate` function.
- If you call the `CFI_allocate` or `CFI_deallocate` functions on a C descriptor, the allocation status of the Fortran variable that the C descriptor describes and the allocation status of any associated allocatable variable change accordingly.

Use of the C descriptor

If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument, obey the following rules:
- If the corresponding dummy argument of the C descriptor in the Fortran interface has the **INTENT(IN)** attribute, you cannot make any changes to the contents or location of the C descriptor, including establishment and update.
- If the corresponding dummy argument of the C descriptor in the Fortran interface has both of the **POINTER** and **INTENT(OUT)** attributes, you can access the `base_addr` member of the C descriptor only after you change the value of it.

Argument association

The following rules apply to argument association:

- If a dummy argument in an interoperable interface is allocatable or is a pointer, the corresponding C formal parameter is interpreted as the address of a C descriptor.
- If the actual argument is an array and the dummy argument is allocatable or is a pointer, the bounds of the dummy argument are assumed from the actual argument.
- When a Fortran procedure with an allocatable dummy argument of **INTENT(OUT)** attribute is called by a C function, and the actual argument in the C function is the address of a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure.
- When a C function is called by a Fortran procedure from an interface with an allocatable dummy argument of **INTENT(OUT)** attribute, and the actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation; in other words, the variable is deallocated before the execution of the C function begins.

Lifetimes

Be watchful of the following rules about the lifetime of interoperable allocatable and pointer arguments:

- If the lifetime of a C descriptor for an allocatable object that was established by C ends before the program exits, the object must be deallocated before the lifetime of its C descriptor ended.
- If a Fortran pointer is associated with a C object, the association status of the Fortran pointer becomes undefined when the lifetime of the C object ends.
- All the C descriptors and C pointers that are associated to any part of a Fortran object become undefined, and any further use of the C descriptors and C pointers becomes undefined behavior in any of the following circumstances:
  - The Fortran object is deallocated.
  - The execution of host instance of the Fortran object is completed.
  - The association status of the Fortran object becomes undefined.

Value of the member fields of the C descriptor

The following rules apply to the value of the member fields of the C descriptor:

- If the C descriptor represents a deallocated allocatable variable or a disassociated pointer, the value of the `base_addr` field is null pointer.
- If the C descriptor represents an array pointer or allocatable array, the `lower_bound` value of each element in the `dim` member of the descriptor is determined by argument association, allocation, or pointer association.

Related information
- "ALLOCATABLE (Fortran 2003)" on page 285
- "INTENT" on page 403
- "POINTER (Fortran 90)" on page 426

Example: Allocatable and pointer arguments

This example shows how to use allocatable array and array pointer arguments in interoperable procedures to multiply the following matrixes. One procedure is defined on the Fortran side and is called from the C side.

$$\begin{vmatrix} 3, & 5 \\ 6, & 7 \\ -1, & 2 \end{vmatrix} X \begin{vmatrix} 1 & 0 \\ 2 & 1 \end{vmatrix}$$

On the Fortran side, the matrix_mult_fortran procedure that multiplies the matrixes is defined as follows.

```fortran
SOUBROUTINE matrix_mult_fortran(res, a, b) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING
  REAL(C_DOUBLE), ALLOCATABLE, INTENT(OUT) :: res(:,:) !Allocatable array
  REAL(C_DOUBLE), POINTER, INTENT(IN) :: a(:,:)        !Array pointer
  REAL(C_DOUBLE), POINTER, INTENT(IN) :: b(:,:)        !Array pointer
  INTEGER(8) :: ub1, ub2

  ub1 = UBOUND(a,1) - LBOUND(a,1)+1
  ub2 = UBOUND(b,2) - LBOUND(b,2)+1
  ALLOCATE(res(ub1, ub2))
  res = MATMUL(a, b)
END
```

On the C side, the matrix_mult_fortran procedure that is defined on the Fortran side is first declared and called in the main function.

```c
#include <stdio.h>
#include <assert.h>
#include "ISO_Fortran_binding.h"
// Prototype for the function that is defined on the Fortran side.
void matrix_mult_fortran(CFI_cdesc_t * res, CFI_cdesc_t * a, CFI_cdesc_t * b);

// Function to print a Fortran array described by a CFI descriptor
void print2DFortranArrayOfDoubles(CFI_cdesc_t * d);

int main()
{
  CFI_CDESC_T(2) a, b, res;
  CFI_cdesc_t * desc_a = (CFI_cdesc_t *) &a;
  CFI_cdesc_t * desc_b = (CFI_cdesc_t *) &b;
  CFI_cdesc_t * desc_res = (CFI_cdesc_t *) &res;
  // Because Fortran is column major and C is row major, the input matrixes
  // defined on the C side are transposed.
  double matrixa[2][3] = { {3, 6, -1}, {5, 7, 2} };
  double matrixb[2][2] = { {1, 2}, {0, 1} };
  const CFI_index_t a_extents[2] = { 3, 2 };
  const CFI_index_t b_extents[2] = { 2, 2 };
  int rc;
  rc = CFI_establish(desc_a,
      matrixa,
      CFI_attribute_pointer,
      CFI_type_double,
      sizeof(double),
      (CFI_rank_t)2,
      a_extents);
  // Make sure that the result is successful.
  assert(CFI_SUCCESS == rc);

  rc = CFI_establish(desc_b,
      matrixb,
      CFI_attribute_pointer,
      CFI_type_double,
      sizeof(double),
      (CFI_rank_t)2,
      b_extents);
```

```
    assert(CFI_SUCCESS == rc);

  rc = CFI_establish(desc_res,
       NULL,
       CFI_attribute_allocatable,
       CFI_type_double,
       sizeof(double),
       (CFI_rank_t)2,
       NULL);
  assert(CFI_SUCCESS == rc);

  // Call the Fortran procedure to multiply the matrix.
  matrix_mult_fortran(desc_res, desc_a, desc_b);

  assert(desc_res->base_addr);

  // Call the C function to print the result.
  print2DFortranArrayOfDoubles(desc_res);

  rc = CFI_deallocate(desc_res);
  assert(CFI_SUCCESS == rc);
  return 0;
}
```

The `print2DFortranArrayOfDoubles` function that regulates the display of the matrix is also defined on the C side.

```
void print2DFortranArrayOfDoubles(CFI_cdesc_t * d)
{
  CFI_index_t extents[2];
  CFI_index_t strides[2];
  int i, j;
  char * prow;

  prow = (char *)d -> base_addr;
  extents[0] = d->dim[0].extent;
  extents[1] = d->dim[1].extent;
  strides[0] = d->dim[0].sm;
  strides[1] = d->dim[1].sm;

  // Print the values of the array one row at a time.
  // Note: This is not cache friendly, because Fortran arrays are column major.

  for(i = 0; i < extents[0]; i++)
  {
    char * tp = prow;
    for(j = 0; j < extents[1]; j++)
    {
      printf("%f ", *(double *)tp);
      tp += strides[1];
    }
    prow += strides[0];
    printf("\n");
  }
}
```

The output is as follows:

```
13.0 5.0
20.0 7.0
3.0 2.0
```

Allocatable and pointer arguments in parallel environments

When you use interoperable procedures with allocatable and pointer arguments in parallel environments, pay attention to the following instances:

- When an actual argument is associated with an allocatable or pointer dummy argument, XL Fortran might create temporary objects to convert a C descriptor to an internal format or vice versa. These temporary copies might be created locally, and each thread might have its own copy of the descriptor. Synchronize read and write operations that affect descriptor fields before you proceed.
- When an allocatable or pointer entity is passed from a non-**BIND(C)** procedure to a **BIND(C)** procedure to be allocated or to associate the pointer, the updates to the descriptor for that entity will become visible to the non-**BIND(C)** procedure only after the execution of the **BIND(C)** procedure is completed. Similarly, when an allocatable or pointer dummy argument of a **BIND(C)** procedure is passed to a non-**BIND(C)** procedure to be allocated or to associate the pointer, the updates to the descriptor for that entity will become visible to the **BIND(C)** procedure only after the execution of the non-**BIND(C)** procedure is completed.

Related information
- "ALLOCATABLE (Fortran 2003)" on page 285
- "BIND (Fortran 2003)" on page 298
- "POINTER (Fortran 90)" on page 426

Interoperability of assumed-length arguments (TS 29113)

A dummy data object of type **CHARACTER** of an interoperable procedure can have assumed length. This allows passing an entity of type **CHARACTER** with any length between Fortran procedures and C functions through an assumed length dummy argument.

If a procedure whose Fortran interface declares a nonallocatable and nonpointer assumed-length dummy argument is invoked from Fortran, the corresponding effective argument of type CFI_cdesc_t has the rank and extents of those declared in the Fortran interface for the corresponding assumed-length dummy argument. For the *dim* member of the effective argument, the value of the *lower_bound* member of each element is zero.

Example 1: Both caller and callee are written in Fortran

```
PROGRAM sample
  INTERFACE
    SUBROUTINE sub(arg) BIND(C)
      CHARACTER(*) :: arg
      !arg is a nonallocatable and nonpointer dummy argument of
      !type character with assumed-length.
    END SUBROUTINE
  END INTERFACE
  CHARACTER(4) :: c1
  !c1 is a variable of type character with a length of four.
  CALL sub(c1)
END PROGRAM sample

!After the program sample, the body of subroutine sub is defined.
SUBROUTINE sub(arg) BIND(C)
  CHARACTER (*) :: arg
  PRINT*, LEN(arg)
END SUBROUTINE
```

Example 2: The caller is a C procedure and the callee is written in Fortran

```
SUBROUTINE sub(arg) BIND(C)
  CHARACTER(*) :: arg
  PRINT*, "The length of the string is:",LEN(arg)
END SUBROUTINE
```

```
#include <stdio.h>
#include "ISO_Fortran_binding.h"

//The prototype of the function sub is declared.
int sub(CFI_cdesc_t * arg);

int main()
{
  char *c1 = "jack";
  int rc = 0;
  CFI_cdesc_t* a;
  CFI_CDESC_T(0) at;
  a = (CFI_cdesc_t *) &at;
  rc = CFI_establist(a,c1,CFI_attribute_other,CFI_type_signed_char,4,0,0);
  if (rc != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_establish returns : %s\n", __xlf_CFI_strerror(rc));
    return 1;
  }
  else
  {
    // This will print the length in Fortran subroutine
    sub(a);
  }
  return 0;
}
```

The output is as follows:

```
The length of the string is:4
```

Related information

"ALLOCATABLE (Fortran 2003)" on page 285

"BIND (Fortran 2003)" on page 298

"CONTIGUOUS (Fortran 2008)" on page 323

"POINTER (Fortran 90)" on page 426

Type CFI_cdesc_t defined in ISO_Fortran_binding.h header file

CFI_attribute_other

Interoperability of assumed-type arguments (TS 29113)

To facilitate the interoperability with formal parameters of type void* in C procedures, assumed-type arguments are introduced in Fortran. Assumed-type arguments are entities that are declared with the **TYPE(*)** specifier. They are unlimited polymorphic, and their dynamic types and type parameters are assumed from the corresponding actual arguments. See the following example:

```
! example1f.f
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER(C_SHORT)      :: i_short
INTEGER(C_INT)        :: i_int
INTEGER(C_LONG)       :: i_long

INTERFACE
   SUBROUTINE c_func(a, flag) BIND(C)
   USE, INTRINSIC :: ISO_C_BINDING
```

```
      IMPLICIT NONE
      TYPE(*) :: a
      INTEGER(C_INT), VALUE :: flag
      END SUBROUTINE c_func
END INTERFACE

i_short = 2
i_int = 4
i_long = 4

CALL c_func(i_short, 1)
CALL c_func(i_int, 2)
CALL c_func(i_long, 3)

END
```

```
// example1c.c
#include <stdio.h>
void c_func(void* a, int flag) {
   if(flag == 1)
      printf("Type is c_short %hd\n",*(short*)a);
   else if(flag == 2)
      printf("Type is c_int %d\n",*(int*)a);
   else if(flag == 3)
      printf("Type is c_long %ld\n",*(long*)a);
   return;
}
```

The output of the example is as follows:

```
Type is c_short 2
Type is c_int 4
Type is c_long 4
```

In this example, function c_func is defined in the C program, and it is called by a Fortran procedure. Variable a of type void* is a formal parameter of function c_func in the C program. The corresponding dummy argument of a in the Fortran procedure interface is specified by the **TYPE(*)** specifier, so the dummy argument is of an assumed type. The dynamic type of variable a is determined by the actual argument when function c_func is called. As shown, function c_func is called three times with the actual parameters i_short, i_int, and i_long passed in, so the type of variable a is deduced to c_short, c_int, and c_long accordingly.

Note: Fortran procedures cannot determine the actual types of their assumed-type arguments.

Restrictions:
- An assumed-type argument can neither be an explicit-shaped array nor have any of the following attributes:
  - **ALLOCATABLE**
  - **INTENT(OUT)**
  - **POINTER**
  - **VALUE**
- An assumed-type variable name cannot be used in a designator or an expression except in the following cases:
  - As an actual argument that corresponds to a dummy argument of an assumed type
  - As an actual argument that corresponds to the first dummy argument of any of the following intrinsic procedures:
    - C_LOC

- IS_CONTIGUOUS
- LBOUND
- PRESENT
- RANK
- SHAPE
- SIZE
- UBOUND

- An assumed-type actual argument that corresponds to an assumed-rank dummy argument must be of assumed-shape or assumed-rank.
- An assumed-type dummy argument cannot correspond to an actual argument of a derived type that has type parameters, type-bound procedures, or final subroutines.

Interoperability of assumed-rank arguments (TS 29113)

To facilitate the interoperability with C functions that accept arguments of arbitrary rank including scalar arguments, assumed-rank arguments are introduced in Fortran.

Rules

If a C descriptor is for a nonallocatable nonpointer assumed-rank argument, the following rules apply:
- The *attribute* member of the C descriptor is *CFI_attribute_other*.
- The *lower_bound* member of each element of the *dim* member of the C descriptor is zero.

The lower bound of every dimension of a nonallocatable nonpointer assumed-rank argument on the Fortran side is always one.

If a dummy argument in an interoperable interface is assumed-rank, the corresponding C formal parameter is interpreted as the address of a C descriptor.

For an interoperable procedure with a Fortran interface that has an assumed-rank dummy argument with the **CONTIGUOUS** attribute, the associated actual argument can be noncontiguous.
- If the procedure is invoked from Fortran or the procedure is a Fortran procedure, XL Fortran ensures that the assumed-rank dummy argument is contiguous.
- If the procedure is invoked from C and the procedure is a C procedure, the C procedure must handle the noncontiguous arguments that might be passed to it.

Restrictions:
- If a procedure has an assumed-rank dummy argument, the procedure must have an explicit interface.
- If an object is passed to a Fortran procedure as a nonallocatable nonpointer assumed-rank dummy argument, the lifetime of the object cannot end before the return from the procedure call.
- In the following situations, the C descriptor cannot be modified:
  - Its address is a formal parameter that corresponds to a Fortran actual argument and the corresponding dummy argument in the Fortran interface is for a nonallocatable nonpointer assumed-rank argument.

– Its address is a C actual argument that corresponds to a Fortran dummy argument and the corresponding dummy argument in the Fortran interface is for a nonallocatable nonpointer assumed-rank argument.

Examples

This example calls the `do_square()` function from the Fortran side to calculate the square of every second element of each array, such as scalar objects, one-dimensional, and two-dimensional arrays.

```
PROGRAM mysquare

   USE, INTRINSIC :: iso_c_binding
   IMPLICIT NONE

   INTERFACE
      SUBROUTINE do_square(x) BIND(C)
         IMPORT
         INTEGER(C_INT), INTENT(INOUT) :: x(..)
      END
   END INTERFACE

   INTEGER :: i
   INTEGER(C_INT) :: v_scalar = -1
   INTEGER(C_INT), TARGET :: v_1d(10) = [(i, i=1,10,1)]
   INTEGER(C_INT), TARGET :: v_2d(10,3) = RESHAPE([(i, i=1,30,1)], [10,3])
   INTEGER(C_INT), POINTER :: p_1d(:), p_2d(:,:)

   PRINT *, "before value of scalar:", v_scalar
   PRINT *, "before value of vector:", v_1d
   PRINT *, "before value of 2D:", v_2d

   ! Square the scalar value.
   CALL do_square(v_scalar)

   ! Square every second element of each array.
   p_1d => v_1d(::2)
   p_2d => v_2d(::2,::2)
   CALL do_square(p_1d)
   CALL do_square(p_2d)

   PRINT *, "after value of scalar:", v_scalar
   PRINT *, "after value of vector:", v_1d
   PRINT *, "after value of 2D:", v_2d

END
```

The `do_square()` function is defined as follows on the C side:

```
#include <assert.h>
#include "ISO_Fortran_binding.h"

void do_square(CFI_cdesc_t * x)
{
  assert(x->type == CFI_type_int);

  switch (x->rank)
  {
  case 0:
    {
      int v = *(int *)x->base_addr;
      *(int *)x->base_addr = v * v;
    }
    break;

  case 1:
```

```
    {
      char * p = (char *)x->base_addr;
      int i;

      char * tp = p;
      for(i = 0; i < x->dim[0].extent; i++)
      {
        int v = *(int *)tp;
        *(int *)tp = v * v;
        tp += x->dim[0].sm; // Advance to the next element.
      }
    }
    break;

  case 2:
    {
      CFI_index_t extents[2];
      CFI_index_t strides[2];
      int i, j;
      char * p = (char *)x->base_addr;

      extents[0] = x->dim[0].extent;
      extents[1] = x->dim[1].extent;
      strides[0] = x->dim[0].sm;
      strides[1] = x->dim[1].sm;

      for(i = 0; i < extents[1]; i++)
      {
        char * tp = p;
        for(j = 0; j < extents[0]; j++)
        {
          int v = *(int *)tp;
          *(int *)tp = v * v;
          tp += strides[0]; // Advance to the next element in dimension 1.
        }
        p += strides[1]; // Advance to the next element in dimension 2.
      }
    }
    break;

  default:
    assert(0);
    break;
  }

  return;
}
```

The output is as follows:

```
before value of scalar: -1
before value of vector: 1 2 3 4 5 6 7 8 9 10
before value of 2D: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
after value of scalar: 1
after value of vector: 1 2 9 4 25 6 49 8 81 10
after value of 2D: 1 2 9 4 25 6 49 8 81 10 11 12 13 14 15 16 17 18 19 20 441 22 529 24 625 26 729 28 841 30
```

Related information

- "Assumed-rank objects (TS 29113)" on page 86
- "RANK(A) (TS 29113)" on page 688

The ISO_C_BINDING module

The **ISO_C_BINDING** module provides access to named constants that represent kind type parameters of data representations compatible with C types, the derived type **C_PTR** corresponding to any C data pointer type, the derived type **C_FUNPTR** corresponding to any C function pointer type, and six procedures.

Constants for use as kind type parameters

Table 1 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular kind type parameter values is interoperable with a C type if the type and kind type parameter value are listed in the same row as that C type; if the type is character, interoperability also requires that the length type parameter be omitted or be specified by a constant expression whose value is one. A combination of Fortran type and type parameters that is interoperable with a C type listed in the table is also interoperable with any unqualified C type that is compatible with the listed C type.

Table 93. Interoperable Fortran and C types

Fortran Type	Named Constant (kind type parameter)	Value	C Type
INTEGER	C_SIGNED_CHAR	1	signed char
	C_SHORT	2	short
	C_INT	4	int
	C_LONG	8	long
	C_LONG_LONG	8	long long
	C_SIZE_T	8	size_t
	C_INTPTR_T	8	intptr_t
	TS 29113 C_PTRDIFF_T	8	ptrdiff_t
	C_INTMAX_T	8	intmax_t
	C_INT8_T	1	int8_t
	C_INT16_T	2	int16_t
	C_INT32_T	4	int32_t
	C_INT64_T	8	int64_t
	C_INT_LEAST8_T	1	int_least8_t
	C_INT_LEAST16_T	2	int_least16_t
	C_INT_LEAST32_T	4	int_least32_t
	C_INT_LEAST64_T	8	int_least64_t
	C_INT_FAST8_T	1	int_fast8_t
	C_INT_FAST8_T	4	int_fast8_t
	C_INT_FAST16_T	4	int_fast16_t
	C_INT_FAST32_T	4	int_fast32_t
	C_INT_FAST64_T	8	int_fast64_t
REAL	C_FLOAT	4	float

Table 93. Interoperable Fortran and C types (continued)

Fortran Type	Named Constant (kind type parameter)	Value	C Type
	C_DOUBLE	8	double
	C_LONG_DOUBLE	16	long double
	C_FLOAT_COMPLEX	4	float _Complex
	C_DOUBLE_COMPLEX	8	double _Complex
	C_LONG_DOUBLE_COMPLEX	16	long double _Complex
LOGICAL	C_BOOL	1	_Bool
CHARACTER	C_CHAR	1	char

For example, the type integer with a kind type parameter of C_SHORT is interoperable with the C type short or any C type derived (via typedef) from short.

Note:

1. The named constants in the ISO_C_BINDING module are of type INTEGER(4).

2. In order for any Fortran COMPLEX entity to be interoperable with a corresponding C _Complex entity appearing in C code compatible with gcc, the Fortran code must be compiled with **-qfloat=complexgcc**.

3. Fortran **REAL(C_LONG_DOUBLE)** and **COMPLEX(C_LONG_DOUBLE_COMPLEX)** entities are only interoperable with the corresponding C types if the C code is compiled with an option that enables 128-bit long doubles.

4. Fortran integer entities with kind type parameter values of C_LONG_LONG, C_INT64_T, C_INT_LEAST64_T, C_INT_FAST64_T, and C_INTMAX_T are only interoperable with the corresponding C types if the C compiler supports `long long int` types.

Character constants

The following character constants are provided for compatibility with some commonly used C characters that are represented using escape sequences:

Table 94. Fortran named constants and C characters

Fortran Named Constant	Definition	C Character
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORM_FEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

Other constants

The constant **C_NULL_PTR** is of type **C_PTR**; it has the value of a C null data pointer. The constant **C_NULL_FUNPTR** is of type **C_FUNPTR**; it has the value of a C null function pointer.

Types

The type **C_PTR** is interoperable with any C data pointer type. The type **C_FUNPTR** is interoperable with any C function pointer type. They are both derived types with private components.

Procedures

A C procedure argument is often defined in terms of a C address. The **ISO_C_BINDING** module provides the following procedures.

The **C_ASSOCIATED** function is provided so that Fortran programs can compare C addresses. The **C_F_POINTER** subroutine provides a means of associating a Fortran pointer with the target of a C pointer. The **C_F_PROCPOINTER** subroutine provides a means of associating a Fortran procedure pointer with the target of a C function pointer. The **C_FUNLOC** and **C_LOC** functions are provided so that Fortran applications can determine the appropriate value to use with C facilities. F2008▸ The **C_SIZEOF** function is provided so that Fortran programs can get the size of data entities that are interoperable with C objects. ◂F2008

C_ASSOCIATED(C_PTR_1[, C_PTR_2])
Purpose

Indicates the association status of **C_PTR_1**, or whether **C_PTR_1** and **C_ PTR_2** are associated with the same entity.

Class

Inquiry function

Argument type and attributes

C_PTR_1
> Scalar of type **C_PTR** or **C_FUNPTR**.

C_PTR_2
> An optional scalar of the same type as **C_PTR_1**.

Result type and attributes

Default logical

Result value
- If **C_PTR_2** is absent, then the result is false if **C_PTR_1** is a C null pointer; otherwise, it has a value of true.
- If **C_PTR_2** is present, then the result is false if **C_PTR_1** is a C null pointer. Otherwise, the result is true if **C_PTR_1** compares equal to **C_PTR_2**, and false otherwise.

C_F_POINTER(CPTR, FPTR [, SHAPE])
Purpose

Associates a data pointer with the target of a C pointer and specifies its shape.

Class

Subroutine

Argument type and attributes

CPTR An **INTENT(IN)** argument; a scalar and of type **C_PTR**.

FPTR An **INTENT(OUT)** argument that is a pointer.

SHAPE

An optional **INTENT(IN)** argument of type integer and rank one. If present, its size equals the rank of **FPTR**. **SHAPE** must be present if and only if **FPTR** is an array.

Rules

If the value of **CPTR** is the C address of an interoperable data entity, then:
* **FPTR** has type and type parameters that are interoperable with the type of the entity.
* **FPTR** becomes a pointer associated with the target of **CPTR**.
* If **FPTR** is an array, its shape is specified by **SHAPE**, and each lower bound is 1.

Otherwise, the value of **CPTR** will be the result of a reference to **C_LOC** with a noninteroperable argument *X*. *X* (or its target) cannot have been deallocated or have become undefined due to the execution of a **RETURN** or **END** statement since the reference to **C_LOC**. **FPTR** is a nonpolymorphic, scalar pointer with the same type and type parameters as *X*. It becomes pointer-associated with *X* (or its target if *X* is a pointer).

C_F_PROCPOINTER(CPTR, FPTR)
Purpose

Associates a procedure pointer with the target of a C function pointer.

Class

Subroutine

Argument type and attributes

CPTR An **INTENT(IN)** scalar of type **C_FUNPTR**. It must either be the C address of an interoperable procedure or the result of module procedure **C_FUNLOC** in the **ISO_C_BINDING** intrinsic module.

FPTR An **INTENT(OUT)** procedure pointer. If **CPTR** is the C address of an interoperable procedure, the interface of **FPTR** must be interoperable with the prototype that describes the target of **CPTR**; otherwise, the interface of **FPTR** must have the same characteristics as the target of **CPTR**.

Result

FPTR is a pointer associated with the target of **CPTR**.

Related information

"C_FUNLOC(X)" on page 904

C_FUNLOC(X)
Purpose

Returns the C address of **X**.

Class

Inquiry function

Argument type and attributes

X   A procedure. If it is a procedure pointer, it must be associated.

Result type and attributes

Scalar of type C_FUNPTR

Result value

A value of type C_FUNPTR that represents the C address of **X**.

C_LOC(X)
Purpose

Returns the address of **X**.

Class

Inquiry function

Argument type and attributes

X   Must meet the following rules:
- **X** has either the **POINTER** or the **TARGET** attribute.
- **X** is either a variable with interoperable type and kind type parameters, or a nonpolymorphic variable without length type parameters.
- If **X** is allocatable, **X** must be allocated.
- If **X** is a pointer, **X** must be associated.
- `F2008` If **X** is an array, **X** must be a contiguous nonzero-size array. `F2008`
- **X** cannot be a zero-length string.

Result type and attributes

Scalar of type **C_PTR**

Result value

A value of type **C_PTR** that represents the address of **X**.

C_SIZEOF(X) (Fortran 2008)
Purpose

Returns the size of X in bytes.

Class

Inquiry function

Argument type and attributes

X   An interoperable data entity that is not an assumed-size array.

Result type and attributes

Scalar integer of kind **C_SIZE_T**

Result value
- If X is a scalar, the result value is the result of applying the sizeof operator (in C language) to a C object. The type of that C object is interoperable with the type and type parameter of X.
- If X is an array, the result value is the result of applying the sizeof operator (in C language) to a C object, multiplied by the number of elements in X. The type of that C object is interoperable with the type and type parameter of X.

The ISO_Fortran_binding.h header file (TS 29113)

The ISO_Fortran_binding.h header file provides C type definitions, structures, macros, and functions for creating, accessing, and manipulating certain Fortran entities from C.

Rules

The ISO_Fortran_binding.h header file can be included in any order relative to standard C headers.

The ISO_Fortran_binding.h header file can be included multiple times in the same scope.

C source files that include ISO_Fortran_binding.h must not use any names that start with CFI_ but are not defined in the ISO_Fortran_binding.h header file.

C source files that include ISO_Fortran_binding.h must not define any structure names from the ISO_Fortran_binding.h header file as macro names.

A C descriptor can be initialized, updated, or copied only by calling the `CFI_allocate`, `CFI_deallocate`, `CFI_establish`, `CFI_section`, `CFI_select_part`, and `CFI_setpointer` functions.

Accessing the ISO_Fortran_binding.h header file

The ISO_Fortran_binding.h header file is in `/opt/ibm/xlf/15.1.0/include/ISO_Fortran_binding.h`.

To access the header file in your C program, you must compile your program with the **-I** option. For example, to compile `myprogram.c`, enter:
```
xlf myprogram.f -I/opt/ibm/xlf/15.1.0/include
```

You should link using XL Fortran to ensure that the required libraries are linked in:
```
xlf2008 myprogram.o
```

Type definitions and structures

The ISO_Fortran_binding.h header file provides the following C type definitions and structures.

CFI_attribute_t

A type that describes the attribute code of a C descriptor. The valid values are as follows:

Table 95. Valid values for `CFI_attribute_t`

CFI_attribute_pointer	CFI_attribute_allocatable	CFI_attribute_other

CFI_index_t

A signed long integer type that can describe array subscripts.

CFI_rank_t

An unsigned integer type that can represent values in the range 0 to `CFI_MAX_RANK` inclusive.

CFI_type_t

A type that describes the type of the object that is described by a C descriptor.

The valid values are as follows:

Table 96. Valid values for `CFI_type_t`

CFI_type_signed_char	CFI_type_size_t
CFI_type_short	CFI_type_int8_t
CFI_type_int	CFI_type_int16_t
CFI_type_long	CFI_type_int32_t
CFI_type_long_long	CFI_type_int64_t
CFI_type_float	CFI_type_int_least8_t
CFI_type_double	CFI_type_int_least16_t
CFI_type_long_double	CFI_type_int_least32_t
CFI_type_float_Complex	CFI_type_int_least64_t
CFI_type_double_Complex	CFI_type_int_fast8_t
CFI_type_long_double_Complex	CFI_type_int_fast16_t
CFI_type_Bool	CFI_type_int_fast32_t
CFI_type_char	CFI_type_int_fast64_t
CFI_type_cptr	CFI_type_intmax_t
CFI_type_struct	CFI_type_intptr_t
CFI_type_other	CFI_type_ptrdiff_t

CFI_dim_t

A type definition that describes the lower bound, extent, and stride of an array dimension. It contains the following structure members:

`CFI_index_t lower_bound`
> The lower bound of the dimension that is described.

`CFI_index_t extent`
> The number of elements in the dimension that is described.

CFI_index_t sm
> The stride of the dimension that is described. The stride is the number of bytes between the beginnings of successive elements along a dimension.

CFI_cdesc_t

A type definition that describes a C descriptor. It contains the following structure members:

void *base_addr
> The base address of the data object that is described. For deallocated allocatable objects, `base_addr` is `NULL`.

size_t elem_len
> - For scalars: The size in bytes of the data object that is described.
> - For arrays: The size in bytes of one element of the array.

int version
> The version number of the C descriptor. Currently, the only valid value is available by using the `CFI_VERSION` macro.

CFI_attribute_t attribute
> The attribute code of the C descriptor. For the valid values for `attribute`, see Table 95 on page 906.

CFI_type_t type
> The type code of the C descriptor. Describes the type of the object that is described by the C descriptor. For the valid values for `type`, see Table 96 on page 906.

CFI_rank_t rank
> The rank of the object that is described by the C descriptor. Its value must be in the range 0 ≤ rank ≤ CFI_MAX_RANK. A value of 0 indicates that the object is a scalar. Otherwise, the object is an array.

CFI_dim_t dim[]
> An array of size `rank` that describes the lower bound, extent, and stride of each dimension.

There is a reserved area between `rank` and `dim`. The size of the reserved area is 9 words.

Macros

The ISO_Fortran_binding.h header file defines the following macros.

CFI_CDESC_T(*rank*)
> The `CFI_CDESC_T` macro evaluates to an unqualified type of suitable size and alignment to define a C descriptor of rank *rank*. Pointers to variables that are declared using `CFI_CDESC_T` can be cast to `CFI_cdesc_t *`. For more information, see "Examples" on page 908.
>
> Variables that are declared using `CFI_CDESC_T` must not have an initializer. They must be initialized using `CFI_establish` before they are used for the first time.

CFI_MAX_RANK
> The maximum rank that is supported in C descriptors. XL Fortran supports a maximum rank of 20.

CFI_VERSION
> The C descriptor version.

Macros for error indicators

> The macros for error indicators are as follows:

Table 97. Macros for error indicators

CFI_SUCCESS	CFI_INVALID_ATTRIBUTE
CFI_ERROR_BASE_ADDR_NULL	CFI_INVALID_EXTENT
CFI_ERROR_BASE_ADDR_NOT_NULL	CFI_INVALID_DESCRIPTOR
CFI_INVALID_ELEM_LEN	CFI_ERROR_MEM_ALLOCATION
CFI_INVALID_RANK	CFI_ERROR_OUT_OF_BOUNDS
CFI_INVALID_TYPE	

Examples

The following C code defines a as a C descriptor variable of rank 10, and it defines the ca pointer to a. To establish the C descriptor, it passes pointer ca to CFI_establish().

```
CFI_CDESC_T(10) a;
CFI_cdesc_t *ca = (CFI_cdesc_t *) &a;
CFI_establish(ca, NULL, CFI_attribute_allocatable, CFI_type_float, 0, 10, NULL);
```

Functions

The ISO_Fortran_binding.h header file defines these functions: CFI_address, CFI_allocate, CFI_deallocate, CFI_establish, CFI_is_contiguous, CFI_section, CFI_select_part, CFI_setpointer and __xlf_CFI_strerror.

CFI_address
Purpose

Returns the address of the object that is described by a C descriptor.

C prototype

```
void *CFI_address(const CFI_cdesc_t *dv, const CFI_index_t subscripts[]);
```

Formal parameters

dv The address of the C descriptor that describes the object. The object must not be a deallocated allocatable variable or an unassociated pointer.

subscripts
> If the object is an array of rank r, **subscripts** is the address of an array that contains at least r elements. The values of the elements in **subscripts** array must be within the corresponding bounds of the object that are specified in the C descriptor.
>
> If the object is a scalar, **subscripts** is ignored.

Result value

If the object is an array of rank r, the result is the C address of the array element whose subscripts are the first r elements of the array that is specified by **subscripts**.

If the object is a scalar, the result is the C address of the object.

Examples

The following Fortran program passes array a to function `print_fifth` that is defined on the C side by using a C descriptor:

```
USE ISO_C_BINDING
IMPLICIT NONE
INTERFACE
  SUBROUTINE print_fifth(ca) BIND(C)
    IMPORT C_INT
    INTEGER(C_INT), ALLOCATABLE :: ca(:)
  END SUBROUTINE
END INTERFACE
INTEGER i
INTEGER, ALLOCATABLE :: a(:)
ALLOCATE(a(5:14), SOURCE=[(i, i=1, 10)])
PRINT *, a
CALL print_fifth(a)
END
```

The address of the fifth element of the array can be calculated by adding the lower bound of the C descriptor to the target subscript, 5, and subtracting 1:

```
#include <stdio.h>
#include "ISO_Fortran_binding.h"

void print_fifth(CFI_cdesc_t *ca)
{
  CFI_index_t subscripts_a[1] = {ca->dim[0].lower_bound + 5 - 1};
  int *fifth_a = (int *) CFI_address(ca, subscripts_a);
  printf("%d\n", *fifth_a);
}
```

The output is as follows:

```
1 2 3 4 5 6 7 8 9 10
5
```

CFI_allocate
Purpose

Allocates memory for an allocatable or pointer object that is described by C descriptor **dv**. If the object is an array, the first **dv**->rank elements of `lower_bounds` and **upper_bounds** specify lower bounds and upper bounds for the corresponding dimension. If the object is of type character, `elem_len` is used to describe the element length in the descriptor.

C prototype

```
int CFI_allocate(CFI_cdesc_t *dv,
                 const CFI_index_t lower_bounds[],
                 const CFI_index_t upper_bounds[],
                 size_t elem_len);
```

Formal parameters

dv   The address of the C descriptor that describes the object to be allocated.

`lower_bounds`
>   The address of an array with at least **dv**->rank elements. If **dv**->rank is zero, `lower_bounds` is ignored.

`upper_bounds`
>   The address of an array with at least **dv**->rank elements. If **dv**->rank is zero, **upper_bounds** is ignored.

elem_len
> If the object to be allocated is a character object, **elem_len** is the size of an element of the object in bytes. Otherwise, **elem_len** is ignored.

Result value

If allocation is successful, CFI_SUCCESS is returned. If allocation is unsuccessful, **dv** is not modified and the result is an error indicator. For more information about error indicators, see Macros for error indicators.

Examples

In the following program, a main program that is written in Fortran allocates a two-dimensional array by calling a C function to do the allocation. The code on the Fortran side is as follows:

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT
INTERFACE
  SUBROUTINE ALLOCATE_IN_C(ca) BIND(C)
    IMPORT C_INT
    INTEGER(C_INT), ALLOCATABLE :: ca(:, :)
  END SUBROUTINE
END INTERFACE
INTEGER(C_INT), ALLOCATABLE :: a(:, :)
CALL ALLOCATE_IN_C(a)
PRINT *, ALLOCATED(a)
IF (ALLOCATED(a)) THEN
  PRINT *, LBOUND(a)
  PRINT *, UBOUND(a)
ENDIF
END
```

The code on the C side is as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include "ISO_Fortran_binding.h"
void allocate_in_c(CFI_cdesc_t *ca)
{
  int ind;
  CFI_index_t lower_bounds[] = {1, 2};
  CFI_index_t upper_bounds[] = {3, 4};
  ind = CFI_allocate(ca, lower_bounds, upper_bounds, 0);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_allocate: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }
}
```

The output is as follows:

```
T
1 2
3 4
```

CFI_deallocate
Purpose

Deallocates memory for an object that is described by C descriptor **dv**, and sets **dv**->base_addr to the null pointer.

C prototype

```
int CFI_deallocate(CFI_cdesc_t *dv);
```

Formal parameters

dv The address of the C descriptor that describes the object to be deallocated. If **dv** describes an allocatable object, the object must be allocated. If **dv** describes a pointer object, the object must be associated with a target .

Result value

If deallocation is successful, `CFI_SUCCESS` is returned. If deallocation is unsuccessful, **dv** is not modified and the result is an error indicator. For more information about error indicators, see Macros for error indicators.

CFI_establish
Purpose

Establishes a C descriptor. Updates *__dv__ to be an established C descriptor for a nonallocatable nonpointer data object of known shape, a deallocated allocatable object, or a data pointer.

Rules

If **base_addr** is not a null pointer, the C descriptor with address **dv** describes a nonallocatable entity that is a scalar or a contiguous array. Otherwise, the C descriptor with address **dv** describes a deallocated allocatable object or a disassociated pointer, or the C descriptor with address **dv** has the attribute `CFI_attribute_other` but does not describe a data object.

If the value of **attribute** is `CFI_attribute_pointer` or `CFI_attribute_other`, the lower bounds of the C descriptor with address **dv** are set to zero.

C prototype

```
int CFI_establish(CFI_cdesc_t *dv,
                  void *base_addr,
                  CFI_attribute_t attribute,
                  CFI_type_t type,
                  size_t elem_len,
                  CFI_rank_t rank,
                  const CFI_index_t extents[]);
```

Formal parameters

dv The address of a C object that is large enough to hold a C descriptor of the rank that is specified by **rank**.

It must not have the same value as a C formal parameter that corresponds to a Fortran actual argument. It must not be the address of a C descriptor that describes an allocated allocatable object.

base_addr
A null pointer or the base address of the object to be described.

If **base_addr** is not a null pointer, it must be the address of a contiguous and properly aligned storage sequence for an object of the type that is specified by **type**.

If **base_addr** is the C address of a Fortran data object, **type** and **elem_len** must be consistent with the type and type parameters of the Fortran data object.

attribute

> An attribute code. For the valid values for **attribute**, see Table 95 on page 906.
>
> If **attribute** is CFI_attribute_allocatable, **base_addr** must be a null pointer.

type

> A type code. For the valid values for **type**, see Table 96 on page 906.

elem_len

> If **type** is CFI_type_struct, CFI_type_other, or a Fortran character type, **elem_len** must be greater than zero and equal to the storage size in bytes of an element of the object to be described by the C descriptor with address **dv**.
>
> Otherwise, **elem_len** is ignored.

rank

> The rank of the object to be described by the C descriptor with address **dv**. Its value must be in the range 0 ≤ **rank** ≤ CFI_MAX_RANK.

extents

> If **rank** is greater than zero and **base_addr** is not a null pointer, the following rules apply:
>
> - **extents** is the address of an array with at least **rank** elements.
> - **extents** describes the corresponding extents of the array to be described by the C descriptor with address **dv**.
>
> The extents that are specified by **extents** must not be negative.
>
> If **rank** is zero, or **base_addr** is a null pointer, **extents** is ignored.

Result value

If successful, the result is CFI_SUCCESS. If unsuccessful, the result is an error indicator, and the C descriptor with address **dv** is not modified. For more information about error indicators, see Macros for error indicators.

Examples

In the following C program, a C descriptor is created for an allocatable array of rank 1, and the array is allocated with lower bound 2 and upper bound 4. Then, the array is deallocated.

```
#include <stdlib.h>
#include <stdio.h>
#include "ISO_Fortran_binding.h"

int main()
{
  int ind;
  CFI_CDESC_T(1) a;
  CFI_index_t lower_bounds[1] = {2};
  CFI_index_t upper_bounds[1] = {4};
  CFI_cdesc_t *ca = (CFI_cdesc_t *) &a;

  ind = CFI_establish(ca, NULL, CFI_attribute_allocatable,
                      CFI_type_float, 0, 1, NULL);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_establish: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }

  ind = CFI_allocate(ca, lower_bounds, upper_bounds, 0);
  if (ind != CFI_SUCCESS)
```

```
  {
    fprintf(stderr, "CFI_allocate: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }

  /* ... */

  ind = CFI_deallocate(ca);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_deallocate: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }

  return 0;
}
```

The C program has the same function as the following Fortran program:

```
USE, INTRINSIC :: ISC_C_BINGDING, ONLY: C_FLOAT
USE XLF_POSIX_BINDINGS, ONLY: EXIT
IMPLICIT NONE
INTEGER ind
REAL(C_FLOAT), ALLOCATABLE :: a(:)

ALLOCATE(a(2:4), stat=ind)
IF (ind /= 0) CALL EXIT(ind)

! ...

DEALLOCATE(a, stat=ind)
IF (ind /= 0) CALL EXIT(ind)
END
```

CFI_is_contiguous
Purpose

Tests the contiguity of an array that is described by a C descriptor.

An array is contiguous if the C descriptor that describes the array meets any of the following conditions:
- The value of `attribute` is `CFI_attribute_allocatable`.
- The value of `extent` of the final `dim` element is -1.

C prototype
```
int CFI_is_contiguous(const CFI_cdesc_t *dv);
```

Formal parameters

dv The address of the C descriptor that describes the array.

Result value

If the array described by **dv** is contiguous, the result is 1. Otherwise, the result is 0.

CFI_section
Purpose

Updates the C descriptor with address **result** to describe a section of the array that is described by the C descriptor with address **source**. The lower bounds in the result C descriptor are not changed.

C prototype

```
int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                const CFI_index_t lower_bounds[],
                const CFI_index_t upper_bounds[],
                const CFI_index_t strides[]);
```

Formal parameters

result

The address of a C descriptor in which **result**->rank is equal to **source**->rank minus the number of zero strides. **result**->attribute must be CFI_attribute_other or CFI_attribute_pointer.

If **result** is a C formal parameter that corresponds to a Fortran actual argument, the value of **result**->attribute must be CFI_attribute_pointer.

source

The address of a C descriptor that describes a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer. **source**->elem_len and **result**->elem_len must have the same value. **source**->type and **result**->type must have the same value.

lower_bounds

A null pointer, or the address of an array that specifies the subscripts of an element in the C descriptor with address **source**. The element is the first element of the array section in the Fortran array element order.

If **lower_bounds** is a null pointer, the subscripts of the first element of the array that is described by the C descriptor with address **source** are used. Otherwise, **lower_bounds** must have at least **source**->rank elements.

upper_bounds

A null pointer, or the address of an array that specifies the subscripts of an element in the C descriptor with address **source**. The element is the last element of the array section in the Fortran array element order.

If **upper_bounds** is a null pointer, the C descriptor with address **source** must not describe an assumed-size array, and the subscripts of the last element of the array described by the C descriptor with address **source** are used. Otherwise, **upper_bounds** must have at least **source**->rank elements.

strides

A null pointer, or the address of an array that specifies the strides of the array section in units of elements of the C descriptor with address **source**.

If a stride is 0, the section subscript for the dimension is a subscript and the corresponding elements of **lower_bounds** and **upper_bounds** must have the same value. If **strides** is a null pointer, the stride for each dimension is 1. Otherwise, **strides** must have at least **source**->rank elements.

Result value

If successful, the result is CFI_SUCCESS. If unsuccessful, the result is an error indicator, and the C descriptor with address **result** is not modified. For more information about error indicators, see Macros for error indicators.

Examples

Assume that source is the address of a C descriptor that corresponds to a Fortran array, complex_array. complex_array is declared and allocated on the Fortran side as follows:

```
COMPLEX, ALLOCATABLE :: complex_array(:)
ALLOCATE(complex_array(100))
```

The following C code establishes a C descriptor, `csection`, and updates it to describe the array section `complex_array(20:90:10)`.

```
#include <stdlib.h>
#include <stdio.h>
#include "ISO_Fortran_binding.h"

void complex_section(CFI_cdesc_t *source)
{
  int ind;
  CFI_index_t lower_bounds[] = {20 - 1};
  CFI_index_t upper_bounds[] = {90 - 1};
  CFI_index_t strides[] = {10};
  CFI_CDESC_T(1) section;
  CFI_cdesc_t *csection = (CFI_cdesc_t *) &section;
  ind = CFI_establish(csection, NULL, CFI_attribute_other,
                      CFI_type_float_Complex, 0, 1, NULL);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_establish: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }

  ind = CFI_section(csection, source, lower_bounds, upper_bounds, strides);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_section: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }
}
```

CFI_select_part
Purpose

Updates the C descriptor with address **result** for an array section, each element of which is a part of the corresponding element of an array that is described by the C descriptor with address **source**. The part can be a component of a structure, a substring, or the real or imaginary part of a complex value.

C prototype

```
int CFI_select_part(CFI_cdesc_t *result,
                    const CFI_cdesc_t *source,
                    size_t displacement,
                    size_t elem_len);
```

Formal parameters

result
> The address of a C descriptor that describes the array section. **result**->type specifies the type of the array section. **result**->attribute must be specified as CFI_attribute_other or CFI_attribute_pointer.
>
> If **result** is a C formal parameter that corresponds to a Fortran actual argument, **result**->attribute must be specified as CFI_attribute_pointer.
>
> **result**->rank must have the same value as **source**->rank.

source
> The address of a C descriptor for a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer.

displacement

    The value in bytes to be added to **source**->base_addr to define the base address of the array section. The resulting base address must be appropriately aligned for an object of the specified type. Its value must be in the range 0 ≤ **displacement** ≤ **source**->elem_len - 1.

elem_len

    If **result**->type is a character type, **elem_len** is the storage size in bytes of an element of the character object that is described by the C descriptor with address **result**. Its value must be in the range 1 ≤ **elem_len** ≤ **source**->elem_len.

    If **result**->type is not a character type, **elem_len** is ignored.

Result value

If successful, the result is CFI_SUCCESS. If unsuccessful, the result is an error indicator, and the C descriptor with address **result** is not modified. For more information about error indicators, see Macros for error indicators.

Examples

Assume that source is the address of a C descriptor that corresponds to an allocatable Fortran array, complex_array. complex_array is declared and allocated on the Fortran side as follows:

```
COMPLEX, ALLOCATABLE :: complex_array(:)
ALLOCATE(complex_array(100))
```

The following C code establishes a C descriptor, cimag, and updates it to describe the array section complex_array%im.

```
#include <stdlib.h>
#include <stdio.h>
#include "ISO_Fortran_binding.h"

void complex_select_imag(CFI_cdesc_t *source)
{
  int ind;
  CFI_CDESC_T(1) imag;
  CFI_cdesc_t *cimag = (CFI_cdesc_t *) &imag;
  ind = CFI_establish(cimag, NULL, CFI_attribute_pointer, CFI_type_float, 0, 1, NULL);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_establish: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }

  ind = CFI_select_part(cimag, source, sizeof(float), 0);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_select_part: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
  }
}
```

CFI_setpointer
Purpose

Updates a C descriptor for a Fortran pointer to be associated with a whole object or to be disassociated.

If **source** is a null pointer or **source** describes a disassociated pointer, the C
descriptor with address **result** is updated to describe a disassociated pointer.
Otherwise, the C descriptor with address **result** is updated to describe the object
that is described by the C descriptor with address **source**; the only exception is
that if the rank is greater than zero and **lower_bounds** is not a null pointer, the
lower bounds are replaced by the values of **lower_bounds**.

C prototype

```
int CFI_setpointer(CFI_cdesc_t *result,
                   CFI_cdesc_t *source,
                   const CFI_index_t lower_bounds[]);
```

Formal parameters

result
>   The address of a C descriptor that describes a pointer.

source
>   A null pointer or the address of the C descriptor for a nonallocatable
>   nonpointer object, an allocated allocatable object, or a data pointer object.
>
>   If **source** is not a null pointer, the following rules apply:
>   - **source->**elem_len must have the same value as **result->**elem_len, and
>   - **source->**rank must have the same value as **result->**rank, and
>   - **source->**type must have the same value as **result->**type.

lower_bounds
>   If **source** is a null pointer, or **source->**rank is zero, **lower_bounds** is ignored.
>   Otherwise, **source** is the address of an array with a least **source->**rank
>   elements. The elements of **lower_bounds** provide the lower bounds for each
>   corresponding dimension of the C descriptor with address **result**. The extents
>   and memory strides are copied from the C descriptor with address **source**.

Result value

If successful, the result is CFI_SUCCESS. If unsuccessful, the result is an error
indicator, and the C descriptor with address **result** is not modified. For more
information about error indicators, see Macros for error indicators.

Examples

In the following C program, function func takes a C descriptor pointer formal
parameter. If the C descriptor corresponds to an allocatable array actual argument,
every call to CFI_address must take the lower bound of each dimension into
account. To avoid this, function func creates a pointer C descriptor and associates
it with *target, and function func sets the lower bounds of each dimension to
zero.

```
void func(CFI_cdesc_t *target)
{
  int ind;
  CFI_index_t lower_bounds[CFI_MAX_RANK] = {0};
  CFI_CDESC_T(CFI_MAX_RANK) p;
  CFI_cdesc_t *cp = (CFI_cdesc_t *) &p;
  ind = CFI_establish(cp, NULL, CFI_attribute_pointer, target->type,
                      target->elem_len, target->rank, NULL);
  if (ind != CFI_SUCCESS)
  {
    fprintf(stderr, "CFI_establish: %s\n", __xlf_CFI_strerror(ind));
    exit(1);
```

```
    }

    ind = CFI_setpointer(cp, target, lower_bounds);
    if (ind != CFI_SUCCESS)
    {
      fprintf(stderr, "CFI_setpointer: %s\n", __xlf_CFI_strerror(ind));
      exit(1);
    }

    /* You can begin to work with cp that has assumed zero
       lower bounds regardless of the lower bounds in target. */
    /* ... */
  }
```

__xlf_CFI_strerror (IBM extension)
Purpose

Maps numeric error indicators into a message string.

C prototype

```
char *__xlf_CFI_strerror(int ind);
```

Formal parameters

ind
    An error indicator. For more information, see Macros for error indicators.

Result value

The result is a pointer to a message string that names the error that is specified by
ind.

Examples

`__xlf_CFI_strerror(CFI_SUCCESS)` returns the pointer to message string
`"CFI_SUCCESS"`.

Binding labels

A binding label is a value of type default character that specifies the name by
which a variable, common block, or a procedure is known to the C compiler.

If a variable, common block, or non-dummy procedure has the **BIND** attribute that
is specified with the **NAME=** specifier, the binding label is the value of the
expression specified for the **NAME=** specifier. If the **BIND** attribute is not specified
with a **NAME=** specifier, the binding label is the same as the name of the entity
using lower case letters. The case of letters in the binding label is significant, but
leading and trailing blanks are ignored.

If a Fortran entity with the **BIND** attribute has the same binding label as that of a
C entity with external linkage, the Fortran entity is associated with the C entity.
The binding label of a C entity with external linkage is the same as the name of
the C entity.

An entity of the program must not be identified by more than one binding label.

A binding label cannot be the same as another binding label or a name used to identify any global entity of the Fortran program, ignoring differences in case except when **-qmixed** (or **-U**) is specified.

Chapter 19. The ISO_FORTRAN_ENV intrinsic module

The **ISO_FORTRAN_ENV** intrinsic module provides constants and functions relating to the Fortran environment. The kind of the constants in this module, and the value of the **NUMERIC_STORAGE_SIZE** constant assume a default integer size of 4.

ISO_FORTRAN_ENV constants

This section presents the constants of the ISO_FORTRAN_ENV intrinsic module.

CHARACTER_KINDS (Fortran 2008)
Purpose

An array containing the kind type parameter values supported by XL Fortran for entities of character type.

Type

Default integer array of rank 1 and size 1.

Value

[1]

CHARACTER_STORAGE_SIZE
Purpose

The size, expressed in bits, of the character storage unit.

Type

Default integer scalar.

Value

8

ERROR_UNIT
Purpose

Identifies the preconnected external unit used for error reporting.

Type

Default integer scalar.

Value

0

FILE_STORAGE_SIZE

Purpose

The size, expressed in bits, of the file storage unit.

Type

Default integer scalar.

Value

8

INT8 (Fortran 2008)

Purpose

The kind type parameter value for an 8-bit integer.

Type

Default integer scalar.

Value

1

INT16 (Fortran 2008)

Purpose

The kind type parameter value for a 16-bit integer.

Type

Default integer scalar.

Value

2

INT32 (Fortran 2008)

Purpose

The kind type parameter value for a 32-bit integer.

Type

Default integer scalar.

Value

4

INT64 (Fortran 2008)
Purpose

The kind type parameter value for a 64-bit integer.

Type

Default integer scalar.

Value

8

INTEGER_KINDS (Fortran 2008)
Purpose

An array containing the kind type parameter values supported by XL Fortran for entities of integer type.

Type

Default integer array of rank 1 and size 4.

Value

`[ INT8, INT16, INT32, INT64 ]`

INPUT_UNIT
Purpose

Identifies the preconnected external unit used for input.

Type

Default integer scalar.

Value

5

IOSTAT_END
Purpose

Assigned to the variable specified in an **IOSTAT=** specifier if an end-of-file condition occurs during execution of a **READ** statement. You must set the **IOSTAT_END=2003std** runtime option to get this value for end-of-file conditions on internal files. (See the **IOSTAT_END** runtime option in the *XL Fortran Compiler Reference* for more information.)

Type

Default integer scalar.

Value

-1

IOSTAT_EOR
Purpose

Assigned to the variable specified in an **IOSTAT=** specifier if an end-of-record condition occurs during execution of a **READ** statement.

Type

Default integer scalar.

Value

-4

IOSTAT_INQUIRE_INTERNAL_UNIT (Fortran 2008)
Purpose

The IOSTAT value in user-defined derived type input/output when the INQUIRE statement is used with a unit number that identifies an internal file.

Type

Default integer scalar.

Value

238

Example

```
MODULE m
  IMPLICIT NONE
  TYPE dt
    INTEGER, ALLOCATABLE :: i
    CONTAINS
      PROCEDURE :: write_dt
      GENERIC :: WRITE(formatted) => write_dt
  END TYPE

  CONTAINS
    SUBROUTINE write_dt(dtv, unit, iotype, v_list, iostat, iomsg)
      CLASS(dt), INTENT(IN) :: dtv
      INTEGER, INTENT(IN) :: unit
      CHARACTER(*), INTENT(IN) :: iotype
      INTEGER, INTENT(IN) :: v_list(:)
      INTEGER, INTENT(OUT) :: iostat
      CHARACTER(*), INTENT(INOUT) :: iomsg
      INQUIRE(unit, iostat = iostat)
    END SUBROUTINE
END MODULE m

USE, INSTRINSIC :: ISO_FORTRAN_ENV
USE m

IMPLICIT NONE
```

```
                        TYPE(dt) d
                        CHARACTER(10) :: internal_file
                        INTEGER :: iostat

                        WRITE(internal_file, *, iostat = iostat) d
                        PRINT *, (iostat == IOSTAT_INQUIRE_INTERNAL_UNIT)      ! prints t

                        END
```

LOGICAL_KINDS (Fortran 2008)
Purpose

An array containing the kind type parameter values supported by XL Fortran for entities of logical type.

Type

Default integer array of rank 1 and size 4.

Value

`[ INT8, INT16, INT32, INT64 ]`

NUMERIC_STORAGE_SIZE
Purpose

The size, expressed in bits, of the numeric storage unit.

Type

Default integer scalar.

Value

32

OUTPUT_UNIT
Purpose

Identifies the preconnected external unit used for output.

Type

Default integer scalar.

Value

6

REAL32 (Fortran 2008)
Purpose

The kind type parameter value for a 32-bit real.

Type

Default integer scalar.

Value

4

REAL64 (Fortran 2008)
Purpose

The kind type parameter value for a 64-bit real.

Type

Default integer scalar.

Value

8

REAL128 (Fortran 2008)
Purpose

The kind type parameter value for a 128-bit real.

Type

Default integer scalar.

Value

16

REAL_KINDS (Fortran 2008)
Purpose

An array containing the kind type parameter values supported by XL Fortran for entities of real type.

Type

Default integer array of rank 1 and size 3.

Value

```
[ REAL32, REAL64, REAL128 ]
```

ISO_FORTRAN_ENV functions

This section presents the functions of the ISO_FORTRAN_ENV intrinsic module.

COMPILER_OPTIONS (Fortran 2008)

Class

Specification inquiry function.

Argument types and attributes

None.

Result type and attributes

Character scalar.

Result value

The result value contains the compiler options, configuration file, and environment variables that are in effect when the current compilation unit is compiled. The compiler options specified by @PROCESS directives are not included. The result is formatted in the same way as the information obtained with the **-qsaveopt** option.

Example

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
CHARACTER(*), PARAMETER :: options = COMPILER_OPTIONS()

WRITE(output_unit, *, delim = 'quote') options
END
```

Output:

```
"@(#)opt f /opt/ibm/xlf/15.1.0/bin/xlf90 example.f
@(#)cfg -qxlf90=noautodealloc:nosignedzero:oldpad -qfree=f90
-qxlf2003=nopolymorphic:nobozlitargs:nostopexcept:novolatile:noautorealloc:oldnaninf
-bh:4"
```

Related information

 -qsaveopt

COMPILER_VERSION (Fortran 2008)

Class

Specification inquiry function.

Argument types and attributes

None.

Result type and attributes

Character scalar.

Result value

The result value contains the name and version information of the compiler that compiles the current compilation unit.

Example

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
CHARACTER(*), PARAMETER :: version = COMPILER_VERSION()

WRITE(output_unit, *, delim = 'quote') version
END
```

Related information

 -qversion

Chapter 20. Floating-point control and inquiry procedures

XL Fortran provides several ways that allow you to query and control the floating-point status and control register of the processor directly. These include:

- **fpgets** and **fpsets** subroutines
- Efficient floating-point control and inquiry procedures
- IEEE floating-point procedures, as specified in the Fortran 2003 standard

The **fpgets** and **fpsets** subroutines retrieve and set the status of floating-point operations, respectively. Instead of calling operating system routines directly, these subroutines use an array of logicals named **fpstat** to pass information back and forth.

XL Fortran also provides procedures in the `xlf_fp_util` module that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the **fpgets** and **fpsets** subroutines; they are mapped into inlined machine instructions that directly manipulate the floating-point status and control register.

XL Fortran includes the **IEEE_ARITHMETIC**, **IEEE_EXCEPTIONS**, and **IEEE_FEATURES** modules to take advantage of the Fortran 2003 standard rules for the IEEE floating-point status semantics.

If you use the procedures in this chapter to set the floating-point status and control register, you can specify the **-qfloat=fenv** option.

See the **-qfloat=fenv** option in the *XL Fortran Compiler Reference* for more information.

fpgets fpsets

The **fpgets** and **fpsets** subroutines retrieve and set the status of the floating-point operations, respectively. The include file `/usr/include/fpdc.h` contains the data declarations (specification statements) for the two subroutines. The include file `/usr/include/fpdt.h` contains the data initializations (data statements) and must be included in a block data program unit.

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 can be used to specify floating-point rounding modes. See fpgets and fpsets subroutines in the *XL Fortran Optimization and Programming Guide* for examples and information on the elements of the **fpstat** array.

Note: The `XLF_FP_UTIL` intrinsic module provides procedures for manipulating the status of floating-point operations that are more efficient than the **fpgets** and **fpsets** subroutines. For more information, see "Efficient floating-point control and inquiry procedures" on page 930.

Examples

```
CALL fpgets( fpstat )
 ...
CALL fpsets( fpstat )
BLOCK DATA
INCLUDE 'fpdc.h'
INCLUDE 'fpdt.h'
END
```

Efficient floating-point control and inquiry procedures

XL Fortran provides several procedures that allow you to query and control the floating-point status and control register of the processor directly. These procedures are more efficient than the fpgets and fpsets subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the module `xlf_fp_util`, which contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at compile time rather than at link time. You can use the argument names listed in the examples as the names for keyword arguments when calling a procedure. The following files are supplied for the `xlf_fp_util` module:

File name	File type	Locations
xlf_fp_util.mod	module symbol file	• *install path*/xlf/15.1.0/include

To use these procedures, you must add a USE XLF_FP_UTIL statement to your source file. For more information on **USE**, see "USE" on page 483.

If there are name conflicts (for example if the accessing subprogram has an entity with the same name as a module entity), use the **ONLY** clause or the renaming features of the **USE** statement. For example,

```
USE XLF_FP_UTIL, NULL1 => get_fpscr, NULL2 => set_fpscr
```

When compiling with the **-U** option, you must code the names of these procedures in all lowercase. We will show the names in lowercase here as a reminder.

The fpscr procedures are:
• "clr_fpscr_flags" on page 932
• "get_fpscr" on page 932
• "get_fpscr_flags" on page 932
• "get_round_mode" on page 933
• "set_fpscr" on page 934
• "set_fpscr_flags" on page 934
• "set_round_mode" on page 934

The following table lists the constants that are used with the fpscr procedures:

Family	Constant	Description
IEEE Rounding Modes	FP_RND_RN	Round toward nearest (default)
	FP_RND_RZ	Round toward zero
	FP_RND_RP	Round toward plus infinity
	FP_RND_RM	Round toward minus infinity
	FP_RND_MODE	Used to obtain the rounding mode from an FPSCR flags variable or value
IEEE Exception Enable Flags **1**	TRP_INEXACT	Enable inexact trap
	TRP_DIV_BY_ZERO	Enable divide-by-zero trap
	TRP_UNDERFLOW	Enable underflow trap
	TRP_OVERFLOW	Enable overflow trap
	TRP_INVALID	Enable invalid trap
	FP_ENBL_SUMM	Trap enable summary or enable all
IEEE Exception Status Flags	FP_INVALID	Invalid operation exception
	FP_OVERFLOW	Overflow exception
	FP_UNDERFLOW	Underflow exception
	FP_DIV_BY_ZERO	Divide-by-zero exception
	FP_INEXACT	Inexact exception
	FP_ALL_IEEE_XCP	All IEEE exceptions summary flags
	FP_COMMON_IEEE_XCP	All IEEE exceptions summary flags excluding the FP_INEXACT exception
Machine Specific Exception Details Flags	FP_INV_SNAN	Signaling NaN
	FP_INV_ISI	Infinity – Infinity
	FP_INV_IDI	Infinity / Infinity
	FP_INV_ZDZ	0 / 0
	FP_INV_IMZ	Infinity * 0
	FP_INV_CMP	Unordered compare
	FP_INV_SQRT	Square root of negative number
	FP_INV_CVI	Conversion to integer error
	FP_INV_VXSOFT	Software request
Machine Specific Exception Summary Flags	FP_ANY_XCP	Any exception summary flag
	FP_ALL_XCP	All exceptions summary flags
	FP_COMMON_XCP	All exceptions summary flags excluding the FP_INEXACT exception

Notes:

- **1** In order to enable exception trapping, you must set the desired IEEE Exception Enable Flags and,
  - change the mode of the user process to allow floating-point exceptions to generate traps with a call to fp_trap, or,
  - compile your program with the appropriate **–qflttrap** suboption. For more information on the **-qflttrap** compiler option and its suboptions, see the *XL Fortran Compiler Reference*.

xlf_fp_util floating-point procedures

This section lists the efficient floating-point control and inquiry procedures in the `XLF_FP_UTIL` intrinsic module.

clr_fpscr_flags
Type

The clr_fpscr_flags subroutine clears the floating-point status and control register flags you specify in the MASK argument. Flags that you do not specify in MASK remain unaffected. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsic procedures described in "Integer bit model" on page 570.

For more information on the FPSCR constants, see FPSCR constants.

Examples

```
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) MASK

! Clear the overflow and underflow exception flags

MASK=(IOR(FP_OVERFLOW,FP_UNDERFLOW))
CALL clr_fpscr_flags(MASK)
```

For another example of the clr_fpscr_flags subroutine, see "get_fpscr_flags."

get_fpscr
Type

The get_fpscr function returns the current value of the floating-point status and control register (fpscr) of the processor.

Result type and attributes

INTEGER(FPSCR_KIND)

Result value

The current value of the floating-point status and control register (FPSCR) of the processor.

Examples

```
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR

FPSCR=get_fpscr()
```

get_fpscr_flags
Type

The get_fpscr_flags function returns the current state of the floating-point status and control register flags you specify in the MASK argument. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsics described in "Integer bit model" on page 570.

For more information on the FPSCR constants, see FPSCR constants.

Result type and attributes

An INTEGER(FPSCR_KIND)

Result value

The status of the FPSCR flags specified by the MASK argument. If a flag specified in the MASK argument is on, the value for the flag will be returned in the return value. The following example requests the status of the FP_DIV_BY_ZERO and FP_INVALID flags.
• If both flags are on, the return value is IOR(FP_DIV_BY_ZERO, FP_INVALID).
• If only the FP_INVALID flag is on, the return value is FP_INVALID.
• If only the FP_DIV_BY_ZERO flag is on, the return value is FP_DIV_BY_ZERO.
• If neither flag is on, the return value is 0.

Examples
```
USE, INTRINSIC :: XLF_FP_UTIL

! ...

IF (get_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID)) .NE. 0) THEN
  ! Either Divide-by-zero or an invalid operation occurred.

  ! ...

  ! After processing the exception, the exception flags are
  ! cleared.
  CALL clr_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID))
END IF
```

get_round_mode
Type

The get_round_mode function returns the current floating-point rounding mode. The return value will be one of the constants FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM. For more information on the rounding mode constants, see FPSCR constants.

Result type and attributes

An INTEGER(FPSCR_KIND)

Result value

One of the constants FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM.

Examples
```
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE

MODE=get_round_mode()
IF (MODE .EQ. FP_RND_RZ) THEN
! ...
END IF
```

set_fpscr
Type

The set_fpscr function sets the floating-point status and control register (fpscr) of the processor to the value provided in the FPSCR argument, and returns the value of the register before the change.

Argument type and attributes

An INTEGER(FPSCR_KIND)

Result type and attributes

An INTEGER(FPSCR_KIND).

Result value

The value of the register before it was set with set_fpscr.

Examples
```
USE, INTRINSIC :: XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR, OLD_FPSCR

FPSCR=get_fpscr()

! ...  Some changes are made to FPSCR  ...

OLD_FPSCR=set_fpscr(FPSCR)   ! OLD_FPSCR is assigned the value of
                             ! the register before it was
                             ! set with set_fpscr
```

set_fpscr_flags
Type

The set_fpscr_flags subroutine allows you to set the floating-point status and control register flags you specify in the MASK argument. Flags that you do not specify in MASK remain unaffected. MASK must be of type INTEGER(FPSCR_KIND). You can manipulate the MASK using the intrinsics described in "Integer bit model" on page 570.

For more information on the FPSCR constants, see FPSCR constants.

Examples
```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) SAVED_FPSCR

SAVED_FPSCR = get_fpscr()               ! Saves the current value
                                        ! of the fpscr register.
CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                                   ! divide-by-zero.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)     ! Restores fpscr register.
```

set_round_mode
Type

The set_round_mode function sets the current floating-point rounding mode, and returns the rounding mode before the change. You can set the mode to FP_RND_RN, FP_RND_RZ, FP_RND_RP or FP_RND_RM. For more information

on the rounding mode constants, see FPSCR constants.

Argument type and attributes

Integer of kind FPSCR_KIND

Result type and attributes

Integer of kind FPSCR_KIND

Result value

The rounding mode before the change.

Examples

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE

MODE=set_round_mode(FP_RND_RZ)   ! The rounding mode is set to
                                 ! round towards zero.  MODE is
! ...                            ! assigned the previous rounding
                                 ! mode.
MODE=set_round_mode(MODE)        ! The rounding mode is restored.
```

IEEE Modules and support (Fortran 2003)

XL Fortran offers support for IEEE floating–point functionality as specified in the Fortran 2003 standard. The standard defines the **IEEE_EXCEPTIONS** module for exceptions, the **IEEE_ARITHMETIC** module to support IEEE arithmetic, and **IEEE_FEATURES** to specify the IEEE features supported by the compiler.

When using the **IEEE_EXCEPTIONS**, or **IEEE_ARITHMETIC** intrinsic modules, the XL Fortran compiler enforces several Fortran 2003 rules regarding the scope of changes to the floating-point status concerning rounding mode, halting mode, and exception flags. This can impede the performance of programs that use these modules, but do not utilize the new floating-point status semantics. For such programs, the **–qstrictieeemod** compiler option is provided to relax the rules on saving and restoring floating-point status.

Notes:
- XL Fortran Extended Precision floating–point numbers are not in the format suggested by the IEEE standard. As a result, some parts of the modules do not support **REAL(16)**.
- On Linux, IEEE modules generate **SIGFPE** signals.

Compiling and exception handling

XL Fortran provides a number of options for strict compliance with the IEEE standard.
- Use **-qfloat=nomaf** to ensure compatibility with the IEEE standard for floating-point arithmetic (IEEE 754-1985).
- When compiling programs that change the rounding mode, use **-qfloat=rrm**.
- Use **-qfloat=nans** to detect signaling NaN values. Signaling NaN values can only occur if specified in a program.

- Use the **-qstrict=ieeefp** compiler option for strict conformance to the IEEE standard for floating-point arithmetic on programs compiled with an optimization level of **-O3** or higher, **-qhot**, **-qipa**, **-qpdf**, or **-qsmp**.

For more information on IEEE floating–point and specific explanations of the compiler options listed above, see Implementation details of XL Fortran floating-point processing in the *XL Fortran Optimization and Programming Guide*.

General rules for implementing IEEE modules

The **IEEE_ARITHMETIC**, **IEEE_EXCEPTIONS**, and **IEEE_FEATURES** modules are intrinsic, though the types and procedures defined in these modules are not intrinsic.

All functions contained in IEEE modules are pure.

All procedure names are generic and not specific.

The default value for all exception flags is quiet.

By default, exceptions do not cause halting.

Rounding mode defaults towards nearest.

IEEE derived data types and constants

The IEEE modules define the following derived types.

IEEE_FLAG_TYPE
Type

A derived data type defined by the **IEEE_EXCEPTIONS** module that identifies a particular exception flag. The values for **IEEE_FLAG_TYPE** must be one of the following named constants as defined in the **IEEE_EXCEPTIONS** module:

IEEE_OVERFLOW

Occurs when the result for an intrinsic real operation or an assignment has an exponent too large to be represented. This exception also occurs when the real or imaginary part of the result for an intrinsic complex operation or assignment has an exponent too large to be represented.

When using **REAL(4)**, an overflow occurs when the result value's unbiased exponent is > 127 or < –126.

When using **REAL(8)**, an overflow occurs when the result value's unbiased exponent is > 1023 or < –1022.

IEEE_DIVIDE_BY_ZERO

Occurs when a real or complex division has a nonzero numerator and a zero denominator.

IEEE_INVALID

Occurs when a real or complex operation or assignment is invalid.

IEEE_UNDERFLOW

Occurs when the result for an intrinsic real operation or assignment has an absolute value too small to be represented by anything other than zero, and loss of accuracy is detected. The exception also occurs when the real or imaginary part of the result for an intrinsic complex operation or

assignment has an absolute value that is too small to be represented by anything other than zero, and loss of accuracy is detected.

For **REAL(4)**, an underflow occurs when the result has an absolute value $< 2^{-149}$.

For **REAL(8)**, an underflow occurs when the result has an absolute value $< 2^{-1074}$.

IEEE_INEXACT
> Occurs when the result of a real or complex assignment or operation is not exact.

The following constants are arrays of **IEEE_FLAG_TYPE**:

IEEE_USUAL
> An array named constant containing **IEEE_OVERFLOW**, **IEEE_DIVIDE_BY_ZERO**, and **IEEE_INVALID** elements in order.

IEEE_ALL
> An array named constant containing **IEEE_USUAL**, **IEEE_UNDERFLOW**, and **IEEE_INEXACT** elements in order.

IEEE_STATUS_TYPE
Type

A derived data type defined in the **IEEE_ARITHMETIC** module that represents the current floating-point status. The floating-point status encompasses the values of all exception flags, halting, and rounding modes.

IEEE_CLASS_TYPE
Type

A derived data type defined in the **IEEE_ARITHMETIC** module that categorizes a class of floating-point values. The values for **IEEE_CLASS_TYPE** must be one of the following named constants as defined in the **IEEE_ARITHMETIC** module:

IEEE_SIGNALING_NAN	IEEE_POSITIVE_ZERO
IEEE_QUIET_NAN	IEEE_POSITIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_NORMAL
IEEE_NEGATIVE_NORMAL	IEEE_POSITIVE_INF
IEEE_NEGATIVE_DENORMAL	IEEE_OTHER_VALUE
IEEE_NEGATIVE_ZERO	

IEEE_ROUND_TYPE
Type

A derived data type defined in the **IEEE_ARITHMETIC** module that identifies a particular rounding mode. The values for **IEEE_ROUND_TYPE** must be one of the following named constants as defined in the **IEEE_ARITHMETIC** module:

IEEE_NEAREST
> Rounds the exact result to the nearest representable value.

IEEE_TO_ZERO
> Rounds the exact result to the next representable value, towards zero.

IEEE_UP

Rounds the exact result to the next representable value, towards positive infinity.

IEEE_DOWN

Rounds the exact result to the next representable value, towards negative infinity.

IEEE_OTHER

Indicates that the rounding mode does not conform to the IEEE standard.

IEEE_FEATURES_TYPE
Type

A derived data type defined in the **IEEE_FEATURES** module that identifies the IEEE features to use. The values for **IEEE_FEATURES_TYPE** must be one of the following named constants as defined in the **IEEE_FEATURES** module:

IEEE_DATATYPE	IEEE_DATATYPE
IEEE_DENORMAL	IEEE_INVALID_FLAG
IEEE_DIVIDE	IEEE_NAN
IEEE_HALTING	IEEE_ROUNDING
IEEE_INEXACT_FLAG	IEEE_SQRT
IEEE_INF	IEEE_UNDERFLOW_FLAG

IEEE Operators

The **IEEE_ARITHMETIC** module defines two sets of elemental operators for comparing variables of **IEEE_CLASS_TYPE** or **IEEE_ROUND_TYPE**.

==   Allows you to compare two **IEEE_CLASS_TYPE** or two **IEEE_ROUND_TYPE** values. The operator returns true if the values are identical or false if they differ.

/=   Allows you to compare two **IEEE_CLASS_TYPE** or two **IEEE_ROUND_TYPE** values. The operator returns true if the values differ or false if they are identical.

IEEE procedures

To use the following IEEE procedures, you must add a `USE IEEE_ARITHMETIC`, `USE IEEE_EXCEPTIONS`, or `USE IEEE_FEATURES` statement to your source file as required. For more information on the **USE** statement, see "USE" on page 483.

Rules for using IEEE procedures

XL Fortran supports all the named constants in the **IEEE_FEATURES** module.

The **IEEE_ARITHMETIC** module behaves as if it contained a **USE** statement for **IEEE_EXCEPTIONS**. All values that are public in **IEEE_EXCEPTIONS** remain public in **IEEE_ARITHMETIC**.

When the **IEEE_EXCEPTIONS** or the **IEEE_ARITHMETIC** modules are accessible, **IEEE_OVERFLOW** and **IEEE_DIVIDE_BY_ZERO** are supported in the scoping unit for all kinds of real and complex data. To determine the other exceptions supported use the **IEEE_SUPPORT_FLAG** function. Use **IEEE_SUPPORT_HALTING** to determine if halting is supported. Support of other

exceptions is influenced by the accessibility of the named constants **IEEE_INEXACT_FLAG**, **IEEE_INVALID_FLAG**, and **IEEE_UNDERFLOW_FLAG** of the **IEEE_FEATURES** module as follows:

- If a scoping unit has access to **IEEE_UNDERFLOW_FLAG** of **IEEE_FEATURES**, the scoping unit supports underflow and returns true from **IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW, X)**, for **REAL(4)** and **REAL(8)**.
- If **IEEE_INEXACT_FLAG** or **IEEE_INVALID_FLAG** is accessible, the scoping unit supports the exception and returns true from the corresponding inquiry for **REAL(4)** and **REAL(8)**.
- If **IEEE_HALTING** is accessible, the scoping unit supports halting control and returns true from **IEEE_SUPPORT_HALTING(FLAG)** for the flag.

If an exception flag signals on entry to a scoping unit that does not access **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC**, the compiler ensures that the exception flag is signaling on exit. If a flag is quiet on entry to such a scoping unit, it can be signaling on exit.

Further IEEE support is available through the **IEEE_ARITHMETIC** module. Support is influenced by the accessibility of named constants in the **IEEE_FEATURES** module:

- If a scoping unit has access to **IEEE_DATATYPE** of **IEEE_FEATURES**, the scoping unit supports IEEE arithmetic and returns true from **IEEE_SUPPORT_DATATYPE(X)** for **REAL(4)** and **REAL(8)**.
- If **IEEE_DENORMAL**, **IEEE_DIVIDE**, **IEEE_INF**, **IEEE_NAN**, **IEEE_ROUNDING**, or **IEEE_SQRT** is accessible, the scoping unit supports the feature and returns true from the corresponding inquiry function for **REAL(4)** and **REAL(8)**.
- For **IEEE_ROUNDING**, the scoping unit returns true for all the rounding modes **IEEE_NEAREST**, **IEEE_TO_ZERO**, **IEEE_UP**, and **IEEE_DOWN** for **REAL(4)** and **REAL(8)**.

If the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** modules are accessed, and **IEEE_FEATURES** is not, the supported subset of features is the same as if **IEEE_FEATURES** was accessed.

IEEE_CLASS(X)
Type

An elemental IEEE class function. Returns the IEEE class of a floating-point value.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

The result is of type **IEEE_CLASS_TYPE**.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) function must return with a value of true. If you
specify a data type of **REAL(16)**, then **IEEE_SUPPORT_DATATYPE** will return
false, though the appropriate class type will still be returned.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_CLASS_TYPE) :: C
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  C = IEEE_CLASS(X)                 ! C has class IEEE_NEGATIVE_NORMAL
ENDIF
```

IEEE_COPY_SIGN(X, Y)
Type

An elemental IEEE copy sign function. Returns the value of *X* with the sign of *Y*.

Module

IEEE_ARITHMETIC

Syntax

Where *X* and *Y* are of type real, though they may be of different kinds.

Result type and attributes

The result is of the same kind and type as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) and **IEEE_SUPPORT_DATATYPE(Y)** must return
with a value of true.

For supported IEEE special values, such as NaN and infinity, **IEEE_COPY_SIGN**
returns the value of *X* with the sign of *Y*.

IEEE_COPY_SIGN ignores the **–qxlf90=nosignedzero** compiler option.

Note: XL Fortran **REAL(16)** numbers have no signed zero.

Examples

Example 1:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X
DOUBLE PRECISION :: Y
X =  3.0
Y = -2.0
IF (IEEE_SUPPORT_DATATYPE(X) .AND. IEEE_SUPPORT_DATATYPE(Y)) THEN
  X = IEEE_COPY_SIGN(X,Y)           ! X has value -3.0
ENDIF
```

Example 2:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X, Y
Y =  1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)  ! X has value -inf
  X = IEEE_COPY_SIGN(X,Y)               ! X has value +inf
ENDIF
```

IEEE_GET_FLAG(FLAG, FLAG_VALUE)
Type

An elemental IEEE subroutine. Retrieves the status of the exception flag specified. Sets *FLAG_VALUE* to true if the flag is signaling, or false otherwise.

Module

IEEE_ARITHMETIC

Syntax

Where *FLAG* is an **INTENT(IN)** argument of type **IEEE_FLAG_TYPE** specifying the IEEE flag to obtain. *FLAG_VALUE* is an **INTENT(OUT)** default logical argument that contains the value of *FLAG*.

Examples
```
USE, INTRINSIC:: IEEE_EXCEPTIONS
LOGICAL :: FLAG_VALUE
CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE)
IF (FLAG_VALUE) THEN
  PRINT *, "Overflow flag is signaling."
ELSE
  PRINT *, "Overflow flag is quiet."
ENDIF
```

IEEE_GET_HALTING_MODE(FLAG, HALTING)
Type

An elemental IEEE subroutine. Retrieves the halting mode for an exception and sets *HALTING* to true if the exception specified by the flag will cause halting.

Module

IEEE_ARITHMETIC

Syntax

Where *FLAG* is an **INTENT(IN)** argument of type **IEEE_FLAG_TYPE** specifying the IEEE flag. *HALTING* is an **INTENT(OUT)** default logical.

Examples
```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL HALTING
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING)
IF (HALTING) THEN
  PRINT *, "The program will halt on an overflow exception."
ENDIF
```

IEEE_GET_ROUNDING_MODE (ROUND_VALUE)
Type

An IEEE subroutine. Sets *ROUND_VALUE* to the current IEEE rounding mode.

Module

IEEE_ARITHMETIC

Syntax

Where *ROUND_VALUE* is an **INTENT(OUT)** scalar of type **IEEE_ROUND_TYPE**.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
IF (ROUND_VALUE == IEEE_OTHER) THEN
  PRINT *, "You are not using an IEEE rounding mode."
ENDIF
```

IEEE_GET_STATUS(STATUS_VALUE)
Type

An IEEE subroutine. Retrieves the current IEEE floating-point status.

Module

IEEE_ARITHMETIC

Syntax

Where *STATUS_VALUE* is an **INTENT(OUT)** scalar of type **IEEE_STATUS_TYPE**.

Rules

You can only use *STATUS_VALUE* in an **IEEE_SET_STATUS** invocation.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)   ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE)   ! Restore the flags
```

IEEE_GET_UNDERFLOW_MODE(GRADUAL)
Type

An IEEE subroutine. Retrieves the underflow mode in operation.

Type

IEEE_ARITHMETIC

Syntax

Where *GRADUAL* is an **INTENT(OUT)** scalar of type default logical.

Rules

XL Fortran does not support underflow control. Only gradual underflow mode is supported. **IEEE_GET_UNDERFLOW_MODE** always sets *GRADUAL* to true.

IEEE_IS_FINITE(X)
Type

An elemental IEEE function. Tests whether a value is finite. Returns true if **IEEE_CLASS(X)** has one of the following values:

- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_DENORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_POSITIVE_ZERO**
- **IEEE_POSITIVE_DENORMAL**
- **IEEE_POSITIVE_NORMAL**

It returns false otherwise.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

Examples
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, IEEE_IS_FINITE(X)    ! Prints true
ENDIF
```

IEEE_IS_NAN(X)
Type

An elemental IEEE function. Tests whether a value is IEEE Not-a-Number. Returns true if **IEEE_CLASS(X)** has the value **IEEE_SIGNALING_NAN** or **IEEE_QUIET_NAN**. It returns false otherwise.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) and **IEEE_SUPPORT_NAN(X)** must return with a
value of true.

Examples

Example 1:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  IF (IEEE_SUPPORT_SQRT(X)) THEN     ! IEEE-compliant SQRT function
    IF (IEEE_SUPPORT_NAN(X)) THEN
      PRINT *, IEEE_IS_NAN(SQRT(X)) ! Prints true
    ENDIF
  ENDIF
ENDIF
```

Example 2:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_STANDARD(X)) THEN
  PRINT *, IEEE_IS_NAN(SQRT(X))      ! Prints true
ENDIF
```

IEEE_IS_NEGATIVE(X)
Type

An elemental IEEE function. Tests whether a value is negative. Returns true if
IEEE_CLASS(X) has one of the following values:

- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_DENORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_NEGATIVE_INF**

It returns false otherwise.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

Examples
```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.0)) THEN
  PRINT *, IEEE_IS_NEGATIVE(1.0)    ! Prints false
ENDIF
```

IEEE_IS_NORMAL(X)
Type

An elemental IEEE function. Tests whether a value is normal. Returns true if
IEEE_CLASS(X) has one of the following values:
- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_POSITIVE_ZERO**
- **IEEE_POSITIVE_NORMAL**

It returns false otherwise.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

Examples
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  IF (IEEE_SUPPORT_SQRT(X)) THEN     ! IEEE-compliant SQRT function
    PRINT *, IEEE_IS_NORMAL(SQRT(X)) ! Prints false
  ENDIF
ENDIF
```

IEEE_LOGB(X)
Type

An elemental IEEE function. Returns unbiased exponent in the IEEE floating-point format. If the value of *X* is neither zero, infinity, or NaN, the result has the value of the unbiased exponent of *X*, equal to **EXPONENT(X)–1**.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is the same type and kind as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If *X* is zero, the result is negative infinity.

If *X* is infinite, the result is positive infinity.

If *X* is NaN, the result is nan.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  PRINT *, IEEE_LOGB(1.1)  ! Prints 0.0
ENDIF
```

IEEE_NEXT_AFTER(X, Y)
Type

An elemental IEEE function. Returns the next machine-representable neighbor of *X* in the direction towards *Y*.

Module

IEEE_ARITHMETIC

Syntax

Where *X* and *Y* are of type real.

Result type and attributes

Where the result is the same type and kind as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** and **IEEE_SUPPORT_DATATYPE(Y)** must return with a value of true.

If *X* and *Y* are equal the function returns *X* without signaling an exception. If *X* and *Y* are not equal, the function returns the next machine-representable neighbor of *X* in the direction towards *Y*.

The neighbors of zero, of either sign, are both nonzero.

IEEE_OVERFLOW and **IEEE_INEXACT** are signaled when *X* is finite but **IEEE_NEXT_AFTER(X, Y)** is infinite.

IEEE_UNDERFLOW and **IEEE_INEXACT** are signaled when **IEEE_NEXT_AFTER(X, Y)** is denormalized or zero.

If *X* or *Y* is a quiet NaN, the result is one of the input NaN values.

Examples

Example 1:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X = 1.0, Y = 2.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == X + EPSILON(X))  ! Prints true
ENDIF
```

Example 2:
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL(4) :: X = 0.0, Y = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == 2.0**(-149))  ! Prints true
ENDIF
```

IEEE_REM(X, Y)
Type

An elemental IEEE remainder function. The result value, regardless of the rounding mode, is exactly *X–Y*N*, where *N* is the integer nearest to the exact value *X/Y*; whenever $|N - X/Y| = 1/2$, *N* is even.

Module

IEEE_ARITHMETIC

Syntax

Where *X* and *Y* are of type real.

Result type and attributes

Where the result is of type real with the same kind as the argument with greater precision.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) and **IEEE_SUPPORT_DATATYPE(Y)** must return
with a value of true.

If the result value is zero, the sign is the same as *X*.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(4.0)) THEN
  PRINT *, IEEE_REM(4.0,3.0)  ! Prints  1.0
  PRINT *, IEEE_REM(3.0,2.0)  ! Prints -1.0
  PRINT *, IEEE_REM(5.0,2.0)  ! Prints  1.0
ENDIF
```

IEEE_RINT(X)
Type

An elemental IEEE function. Rounds to an integer value according to the current
rounding mode.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real.

Result type and attributes

Where the result is the same type and kind as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If the result has the value zero, the sign is that of *X*.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
  PRINT *, IEEE_RINT(1.1)      ! Prints 1.0
  CALL IEEE_SET_ROUNDING_MODE(IEEE_UP)
  PRINT *, IEEE_RINT(1.1)      ! Prints 2.0
ENDIF
```

IEEE_SCALB(X, I)
Type

An elemental IEEE function. Returns $X * 2^I$.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real and *I* is of type **INTEGER**.

Result type and attributes

Where the result is the same type and kind as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If $X * 2^I$ is representable as a normal number, then the result is a normal number.

If *X* is finite and $X * 2^I$ is too large the **IEEE_OVERFLOW** exception occurs. The result value is infinity with the sign of *X*.

If $X * 2^I$ is too small and there is a loss of accuracy, the **IEEE_UNDERFLOW** exception occurs. The result is the nearest representable number with the sign of *X*.

If *X* is infinite, the result is the same as *X* with no exception signals.

Examples
```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.0)) THEN
  PRINT *, IEEE_SCALB(1.0,2)      ! Prints 4.0
ENDIF
```

IEEE_SELECTED_REAL_KIND([P, R, RADIX])
Type

A transformational IEEE function. Returns a value of the kind type parameter of an IEEE real data type with decimal precision of at least P digits, a decimal exponent range of at least R, `F2008` and a radix of RADIX `F2008` .

Module

IEEE_ARITHMETIC

Syntax

Where *P*, *R*, `F2008` and *RADIX* `F2008` are scalar optional arguments of type integer. At least one argument must be present.

Rules

If P or R is not specified, **SELECTED_REAL_KIND** behaves as if you specified P or R with value 0. If RADIX is not specified, the radix of the selected kind can be any supported value.

The result is the value of the kind type parameter of an IEEE real data type that satisfies the following conditions:
• It has decimal precision, as returned by the **PRECISION** function, of at least P digits.

- It has a decimal exponent range, as returned by the **RANGE** function, of at least R.
- F2008 It has a radix, as returned by the **RADIX** function, of RADIX. F2008

If no such kind type parameter is available, the result has different values depending on different conditions as follows:

- If F2008 the radix is available F2008 , the precision is not available, and the exponent range is available, the result is -1.
- If F2008 the radix is available F2008 , the exponent range is not available, and the precision is available, the result is -2.
- If F2008 the radix is available F2008 , and neither the precision nor the exponent range is available, the result is -3.
- If F2008 the radix is available F2008 , and both the precision and exponent range are available separately but not together, the result is -4.
- F2008 If the radix is not available, the result is -5. F2008

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. However, if several values have the same smallest decimal precision, the smallest value is returned.

F2008 **Currently, the XL Fortran compiler only supports** *RADIX*=2. F2008

Examples

Example 1:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

i = IEEE_SELECTED_REAL_KIND(P = 4, R = 32)
PRINT *, 'IEEESELECTREALKIND(4, 32) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 12, R = 307)
PRINT *, 'IEEESELECTREALKIND(12, 307) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 16, R = 291)
PRINT *, 'IEEESELECTREALKIND(16, 291) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 32, R = 291)
PRINT *, 'IEEESELECTREALKIND(32, 291) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 31, R = 308)
PRINT *, 'IEEESELECTREALKIND(31, 308) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 32, R = 308)
PRINT *, 'IEEESELECTREALKIND(32, 308) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 31, R = 292)
PRINT *, 'IEEESELECTREALKIND(31, 292) = ', i
```

The output of this program is as follows:

```
IEEESELECTREALKIND(4, 32) = 4
IEEESELECTREALKIND(12, 307) = 8
IEEESELECTREALKIND(16, 291) = 16
IEEESELECTREALKIND(32, 291) = -1
IEEESELECTREALKIND(31, 308) = -2
IEEESELECTREALKIND(32, 308) = -3
IEEESELECTREALKIND(31, 292) = -4
```

F2008

Example 2:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

i = IEEE_SELECTED_REAL_KIND(P = 4, R = 32, RADIX = 2)
PRINT *, 'IEEESELECTREALKIND(4, 32, 2) = ', i
```

```
i = IEEE_SELECTED_REAL_KIND(P = 32, R = 308, RADIX = 2)
PRINT *, 'IEEESELECTREALKIND(32, 308, 2) = ', i
i = IEEE_SELECTED_REAL_KIND(P = 31, R = 292, RADIX = 32)
PRINT *, 'IEEESELECTREALKIND(31, 292, 32) = ', i
```

The output of this program is as follows:
```
IEEESELECTREALKIND(4, 32, 2) = 4
IEEESELECTREALKIND(32, 308, 2) = -3
IEEESELECTREALKIND(31, 292, 32) = -5
```

F2008

IEEE_SET_FLAG(FLAG, FLAG_VALUE)
Type

An IEEE subroutine. Assigns a value to an IEEE exception flag.

Module

IEEE_EXCEPTIONS

Syntax

Where *FLAG* is an **INTENT(IN)** scalar or array argument of type
IEEE_FLAG_TYPE corresponding to the value of the flag to be set. *FLAG_VALUE*
is an **INTENT(IN)** scalar or array argument of type logical, corresponding to the
wanted status of the exception flag. The value of *FLAG_VALUE* should be
conformable with the value of *FLAG*.

Rules

If *FLAG_VALUE* is true, the exception flag specified by *FLAG* is set to signaling.
Otherwise, the flag is set to quiet.

Each element of *FLAG* must have a unique value.

Examples
```
USE, INTRINSIC :: IEEE_EXCEPTIONS
CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)
! IEEE_OVERFLOW is now signaling
```

IEEE_SET_HALTING_MODE(FLAG, HALTING)
Type

An IEEE subroutine. Controls continuation or halting after an exception.

Module

IEEE_EXCEPTIONS

Syntax

Where *FLAG* is an **INTENT(IN)** scalar or array argument of type
IEEE_FLAG_TYPE corresponding to the exception flag for which holding applies.
HALTING is an **INTENT(IN)** scalar or array argument of type logical,
corresponding to the wanted halting status. By default exceptions will not cause

halting in XL Fortran. The value of *HALTING* should be conformable with the value of *FLAG*.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If *HALTING* is true, the exception specified by *FLAG* will cause halting. Otherwise, execution will continue after the exception.

Each element of *FLAG* must have a unique value.

Examples

```
@PROCESS FLOAT(NOFOLD)
USE, INTRINSIC :: IEEE_EXCEPTIONS
REAL :: X
CALL IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO, .TRUE.)
X = 1.0 / 0.0
! Program will halt with a divide-by-zero exception
```

IEEE_SET_ROUNDING_MODE (ROUND_VALUE)
Type

An IEEE subroutine. Sets the current rounding mode.

Module

IEEE_ARITHMETIC

Syntax

Where *ROUND_VALUE* is an **INTENT(IN)** argument of type **IEEE_ROUND_TYPE** specifying the rounding mode.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** and **IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)** must return with a value of true.

The compilation unit calling this program must be compiled with the **-qfloat=rrm** compiler option.

All compilation units calling programs compiled with the **-qfloat=rrm** compiler option must also be compiled with this option.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
  PRINT *, IEEE_RINT(1.1)      ! Prints 1.0
  CALL IEEE_SET_ROUNDING_MODE(IEEE_UP)
  PRINT *, IEEE_RINT(1.1)      ! Prints 2.0
ENDIF
```

IEEE_SET_STATUS(STATUS_VALUE)
Type

An IEEE subroutine. Restores the value of the floating-point status.

Module

IEEE_ARITHMETIC

Syntax

Where *STATUS_VALUE* is an **INTENT(IN)** argument of type **IEEE_STATUS_TYPE** specifying the floating-point status.

Rules

STATUS_VALUE must have been set previously by **IEEE_GET_STATUS**.

IEEE_SET_UNDERFLOW_MODE(GRADUAL)
Type

An IEEE subroutine. Sets the current underflow mode.

Module

IEEE_ARITHMETIC

Syntax

Where *GRADUAL* is a scalar argument of type default logical.

Rules

XL Fortran does not support underflow control. Only gradual underflow mode is supported. Calling **IEEE_SET_UNDERFLOW_MODE** with *GRADUAL* set to false has no effect.

IEEE_SUPPORT_DATATYPE or IEEE_SUPPORT_DATATYPE(X)
Type

An inquiry IEEE function. Determines whether the current implementation supports IEEE arithmetic. Support means using an IEEE data format and performing the binary operations of **+**, **-**, and ***** as in the IEEE standard whenever the operands and result all have normal values.

Note: NaN and Infinity are not fully supported for **REAL(16)**. Arithmetic operations do not necessarily propagate these values.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

If *X* is absent, the function returns a value of false.

If *X* is present and **REAL(16)**, the function returns a value of false. Otherwise the function returns true.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)   ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE)   ! Restore the flags
```

IEEE_SUPPORT_DENORMAL or IEEE_SUPPORT_DENORMAL(X) Type

An inquiry IEEE function. Determines whether the current implementation supports denormalized numbers.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

The result has a value of true if the implementation supports arithmetic operations and assignments with denormalized numbers for all arguments of type real where *X* is absent, or for real variables of the same kind type parameter as *X*. Otherwise, the result has a value of false.

IEEE_SUPPORT_DIVIDE or IEEE_SUPPORT_DIVIDE(X) Type

An inquiry IEEE function. Determines whether the current implementation supports division to the accuracy of the IEEE standard.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

The result has a value of true if the implementation supports division with the accuracy specified by the IEEE standard for all arguments of type real where *X* is absent, or for real variables of the same kind type parameter as *X*. Otherwise, the result has a value of false.

IEEE_SUPPORT_FLAG(FLAG) or IEEE_SUPPORT_FLAG(FLAG, X)
Type

An inquiry IEEE function. Determines whether the current implementation supports an exception.

Module

IEEE_EXCEPTIONS

Syntax

Where *FLAG* is a scalar argument of **IEEE_FLAG_TYPE**. *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

The result has a value of true if the implementation supports detection of the exception specified for all arguments of type real where *X* is absent, or for real variables of the same kind type parameter as *X*. Otherwise, the result has a value of false.

If *X* is absent, the result has a value of false.

If *X* is present and of type **REAL(16)**, the result has a value of false. Otherwise the result has a value of true.

IEEE_SUPPORT_HALTING(FLAG)
Type

An inquiry IEEE function. Determines whether the current implementation supports the ability to abort or continue execution after an exception occurs. Support by the current implementation includes the ability to change the halting

mode using **IEEE_SET_HALTING(FLAG)**.

Module

IEEE_EXCEPTIONS

Syntax

Where *FLAG* is an **INTENT(IN)** argument of **IEEE_FLAG_TYPE**.

Result type and attributes

The result is a scalar of type default logical.

Rules

The result returns with a value of true for all flags.

IEEE_SUPPORT_INF or IEEE_SUPPORT_INF(X)
Type

An inquiry IEEE function. Determines whether the current implementation supports IEEE infinity behavior for unary and binary operation. Support indicates that IEEE infinity behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, complies with the IEEE standard.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

The result has a value of true if the implementation supports IEEE positive and negative infinities for all arguments of type real where *X* is absent, or for real variables of the same kind type parameter as *X*. Otherwise, the result has a value of false.

If *X* is of type **REAL(16)**, the result has a value of false. Otherwise the result has a value of true.

IEEE_SUPPORT_IO or IEEE_SUPPORT_IO(X)
Type

An inquiry IEEE function. Determines whether the current implementation supports IEEE base conversion rounding during formatted input/output. Support refers to the ability to do IEEE base conversion during formatted input/output as described in the IEEE standard for the modes **IEEE_UP**, **IEEE_DOWN**, **IEEE_ZERO**, and **IEEE_NEAREST** for all arguments of type real where *X* is absent, or for real variables of the same kind type parameter as *X*.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If *X* is present and of type **REAL(16)**, the result has a value of false. Otherwise, the result returns a value of true.

IEEE_SUPPORT_NAN or IEEE_SUPPORT_NAN(X)
Type

An inquiry IEEE function. Determines whether the current implementation supports the IEEE Not-a-Number facility. Support indicates that IEEE NaN behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, conforms to the IEEE standard.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If *X* is absent, the result has a value of false.

If *X* is present and of type **REAL(16)**, the result has a value of false. Otherwise the result returns a value of true.

IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)
Type

An inquiry IEEE function. Determines whether the current implementation supports a particular rounding mode for arguments of type real. Support indicates the ability to change the rounding mode using **IEEE_SET_ROUNDING_MODE**.

Module

IEEE_ARITHMETIC

Syntax

Where *ROUND_VALUE* is a scalar argument of **IEEE_ROUND_TYPE**. *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

If *X* is absent, the result has a value of true if the implementation supports the rounding mode defined by **ROUND_VALUE** for all arguments of type real. Otherwise, it has a value of false.

If *X* is present, the result returns a value of true if the implementation supports the rounding mode defined by **ROUND_VALUE** for real variables of the same kind type parameter as *X*. Otherwise, the result has a value of false.

If *X* is present and of type **REAL(16)**, the result returns a value of false when **ROUND_VALUE** has a value of **IEEE_NEAREST**. Otherwise the result returns a value of true.

If **ROUND_VALUE** has a value of **IEEE_OTHER** the result has a value of false.

IEEE_SUPPORT_SQRT or IEEE_SUPPORT_SQRT(X)
Type

An inquiry IEEE function. Determines whether the current implementation supports the **SQRT** as defined by the IEEE standard.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) must return with a value of true.

If *X* is absent, the result returns a value of true if **SQRT** adheres to IEEE
conventions for all variables of type **REAL**. Otherwise, the result has a value of
false.

If *X* is present, the result returns a value of true if **SQRT** adheres to IEEE
conventions for all variables of type **REAL** with the same kind type parameter as
X. Otherwise, the result has a value of false.

If *X* is present and of type **REAL(16)**, the result has a value of false. Otherwise the
result returns a value of true.

IEEE_SUPPORT_STANDARD or IEEE_SUPPORT_STANDARD(X)
Type

An inquiry IEEE function. Determines whether all facilities defined in the Fortran
2003 standard are supported.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or array valued argument of type real.

Result type and attributes

The result is a scalar of type default logical.

Rules

If *X* is absent, the result returns a value of false since XL Fortran supports
REAL(16).

If *X* is present, the result returns a value of true if the following functions also
return true:
- **IEEE_SUPPORT_DATATYPE(X)**
- **IEEE_SUPPORT_DENORMAL(X)**
- **IEEE_SUPPORT_DIVIDE(X)**
- **IEEE_SUPPORT_FLAG(FLAG, X)** for every valid flag.
- **IEEE_SUPPORT_HALTING(FLAG)** for every valid flag.
- **IEEE_SUPPORT_INF(X)**
- **IEEE_SUPPORT_NAN(X)**
- **IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X)** for every valid
  ROUND_VALUE

- **IEEE_SUPPORT_SQRT(X)**

Otherwise, the result returns a value of false.

IEEE_SUPPORT_UNDERFLOW_CONTROL() or IEEE_SUPPORT_UNDERFLOW_CONTROL(X)
Type

An inquiry IEEE function. Determines if the ability to control underflow mode during execution is supported.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is a scalar or an array of type real.

Rules

XL Fortran does not support underflow mode control.
IEEE_SUPPORT_UNDERFLOW_CONTROL always returns false.

IEEE_UNORDERED(X, Y)
Type

An elemental IEEE unordered function.

Module

IEEE_ARITHMETIC

Syntax

Where *X* and *Y* are of type real.

Result type and attributes

The result is of type default logical.

Rules

To ensure compliance with the Fortran 2003 standard, the
IEEE_SUPPORT_DATATYPE(X) and **IEEE_SUPPORT_DATATYPE(Y)** must return with a value of true.

Unordered function returns with a value of true if *X* or *Y* is a NaN. Otherwise the function returns with a value of false.

Examples

```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL X, Y
X = 0.0
Y = IEEE_VALUE(Y, IEEE_QUIET_NAN)
PRINT *, IEEE_UNORDERED(X,Y)   ! Prints true
END
```

IEEE_VALUE(X, CLASS)
Type

An elemental IEEE function. Generates an IEEE value as specified by *CLASS*.

Note: Implementation of this function is platform and compiler dependent due to variances in NaN processing on differing platforms. A NaN value saved in a binary file that is read on a different platform than the one that generated the value will have unspecified results.

Module

IEEE_ARITHMETIC

Syntax

Where *X* is of type real. *CLASS* is of type **IEEE_CLASS_TYPE**.

Result type and attributes

The result is of the same type and kind as *X*.

Rules

To ensure compliance with the Fortran 2003 standard, the **IEEE_SUPPORT_DATATYPE(X)** must return with a value of true.

IEEE_SUPPORT_NAN(X) must be true if the value of *CLASS* is **IEEE_SIGNALING_NAN** or **IEEE_QUIET_NAN**.

IEEE_SUPPORT_INF(X) must be true if the value of *CLASS* is **IEEE_NEGATIVE_INF** or **IEEE_POSITIVE_INF**.

IEEE_SUPPORT_DENORMAL(X) must be true if the value of *CLASS* is **IEEE_NEGATIVE_DENORMAL** or **IEEE_POSITIVE_DENORMAL**.

Multiple calls of **IEEE_VALUE(X, CLASS)** return the same result for a particular value of *X*, if kind type parameter and *CLASS* remain the same.

If a compilation unit calls this program with a *CLASS* value of **IEEE_SIGNALING_NAN**, the compilation unit must be compiled with the **–qfloat=nans** compiler option.

CLASS may not have the value **IEEE_OTHER_VALUE**.

Examples
```
USE, INTRINSIC :: IEEE_ARITHMETIC
REAL :: X
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
   X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)
   PRINT *, X ! Prints -inf
END IF
```

Rules for floating-point status

An exception flag set to signaling remains signaling until set to quiet by either the **IEEE_SET_FLAG** or **IEEE_SET_STATUS** subroutines.

The compiler ensures that a call from scoping units using the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** intrinsic modules does not change the floating–point status other than by setting exception flags to signaling.

If a flag is set to signaling on entry into a scoping unit that uses the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** modules, the flag is set to quiet and then restored to signaling when leaving that scoping unit.

In a scoping unit that uses the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** modules, the rounding and halting modes do not change on entry. On return, the rounding and halting modes are the same as on entry.

Evaluating a specification expression can cause an exception to signal.

Exception handlers must not use the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** modules.

The following rules apply to format processing and intrinsic procedures:
• The status of a signaling flag, either signaling or quiet, does not change because of an intermediate calculation that does not affect the result.
• If an intrinsic procedure executes normally, the values of the flags **IEEE_OVERFLOW**, **IEEE_DIVIDE_BY_ZERO**, and **IEEE_INVALID** remain the same on entry to the procedure.
• If a real or complex result is too large for the intrinsic to handle, **IEEE_OVERFLOW** may signal.
• If a real or complex result is a NaN because of an invalid operation, **IEEE_INVALID** may signal.

In a sequence of statements that has no invocations of **IEEE_GET_FLAG**, **IEEE_SET_FLAG**, **IEEE_GET_STATUS**, **IEEE_SET_HALTING**, or **IEEE_SET_STATUS**, the following applies. If the execution of an operation would cause an exception to signal but after execution of the sequence no value of a variable depends on the operation, whether the exception is signaling depends on the optimization level. Optimization transformations may eliminate some code, and thus IEEE exception flags signaled by the eliminated code will not signal.

An exception will not signal if this could arise only during execution of an operation beyond those required or permitted by the standard.

For procedures defined by means other than Fortran, it is the responsibility of the user to preserve floating–point status.

XL Fortran does not always detect floating-point exception conditions for extended precision values. If you turn on floating-point exception trapping in programs that use extended precision, XL Fortran may also generate signals in cases where an exception does not really occur. See Detecting and trapping floating-point exceptions in the *XL Fortran Optimization and Programming Guide* for more information.

Fortran 2003 IEEE derived types, constants, and operators are incompatible with the floating–point and inquiry procedures in **xlf_fp_util**, **fpsets**, and **fpgets** procedures. A value obtained from an IEEE procedure cannot be used in non-IEEE procedures. Within a single scoping unit, do not mix calls to the procedures in **xlf_fp_util**, **fpsets**, and **fpgets** with calls to the IEEE procedures. These procedures

may change the floating–point status when called from scoping units that use the **IEEE_EXCEPTIONS** or **IEEE_ARITHMETIC** modules.

Examples

Example 1: In the following example, the main program calls procedure *P* which uses the **IEEE_ARITHMETIC** module. The procedure changes the floating-point status before returning. The example displays the changes to the floating-point status before calling procedure *P*, on entry into the procedure, on exit from *P*, and after returning from the procedure.

```
PROGRAM MAIN
  USE, INTRINSIC :: IEEE_ARITHMETIC

  INTERFACE
    SUBROUTINE P()
      USE IEEE_ARITHMETIC
    END SUBROUTINE P
  END INTERFACE

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL P()

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_NEAREST) THEN
    PRINT *, "MAIN: ROUNDING MODE: IEEE_NEAREST"
  ENDIF
END PROGRAM MAIN

SUBROUTINE P()
  USE IEEE_ARITHMETIC
  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "   P: FLAGS ON ENTRY: ",FLAG_VALUES

  CALL IEEE_SET_ROUNDING_MODE(IEEE_TO_ZERO)
  CALL IEEE_SET_FLAG(IEEE_UNDERFLOW, .TRUE.)

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_TO_ZERO) THEN
    PRINT *, "   P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO"
  ENDIF
  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "   P: FLAGS ON EXIT: ",FLAG_VALUES
END SUBROUTINE P
```

When using the **–qstrictieeemod** compiler option to ensure compliance with rules for IEEE arithmetic, exception flags set before calling *P* are cleared on entry to *P*. Changes to the floating–point status occurring in *P* are undone when *P* returns, with the exception that flags set in *P* remain set after *P* returns:

```
 MAIN: FLAGS   T F F F F
    P: FLAGS ON ENTRY:  F F F F F
    P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO
    P: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS   T F F T F
 MAIN: ROUNDING MODE: IEEE_NEAREST
```

When the **–qnostrictieeemod** compiler option is in effect, exception flags which
were set before calling *P* remain set on entry to *P*. Changes to the floating-point
status occurring in *P* are propagated to the caller.

```
 MAIN: FLAGS   T F F F F
    P: FLAGS ON ENTRY:  T F F F F
    P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO
    P: FLAGS ON EXIT:  T F F T F
 MAIN: FLAGS   T F F T F
```

Example 2: In the following example, the main program calls procedure *Q* which
uses neither **IEEE_ARITHMETIC** nor **IEEE_EXCEPTIONS**. Procedure *Q* changes
the floating–point status before returning. The example displays the changes to the
floating–point status before calling *Q*, on entry into the procedure, on exit from Q,
and after returning from the procedure.

```
PROGRAM MAIN
  USE, INTRINSIC :: IEEE_ARITHMETIC

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL Q()

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_NEAREST) THEN
    PRINT *, "MAIN: ROUNDING MODE: IEEE_NEAREST"
  ENDIF
END PROGRAM MAIN

SUBROUTINE Q()
  USE XLF_FP_UTIL
  INTERFACE
    FUNCTION GET_FLAGS()
      LOGICAL, DIMENSION(5) :: GET_FLAGS
    END FUNCTION
  END INTERFACE

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  INTEGER(FP_MODE_KIND) :: OLDMODE

  FLAG_VALUES = GET_FLAGS()
  PRINT *, "   Q: FLAGS ON ENTRY: ", FLAG_VALUES

  CALL CLR_FPSCR_FLAGS(FP_OVERFLOW)
  OLDMODE = SET_ROUND_MODE(FP_RND_RZ)
  CALL SET_FPSCR_FLAGS(TRP_OVERFLOW)
  CALL SET_FPSCR_FLAGS(FP_UNDERFLOW)

  IF (GET_ROUND_MODE() == FP_RND_RZ) THEN
    PRINT *, "   Q: ROUNDING MODE ON EXIT: TO_ZERO"
```

```
    ENDIF

  FLAG_VALUES = GET_FLAGS()
  PRINT *, "   Q: FLAGS ON EXIT: ", FLAG_VALUES
END SUBROUTINE Q

! PRINT THE STATUS OF ALL EXCEPTION FLAGS
FUNCTION GET_FLAGS()
  USE XLF_FP_UTIL
  LOGICAL, DIMENSION(5) :: GET_FLAGS
  INTEGER(FPSCR_KIND), DIMENSION(5) :: FLAGS
  INTEGER I

  FLAGS = (/ FP_OVERFLOW, FP_DIV_BY_ZERO, FP_INVALID, &
  &          FP_UNDERFLOW, FP_INEXACT /)
  DO I=1,5
    GET_FLAGS(I) = (GET_FPSCR_FLAGS(FLAGS(I)) /= 0)
  END DO
END FUNCTION
```

When using the **–qstrictieeemod** compiler option to ensure compliance with rules for IEEE arithmetic, exception flags set before *Q* remain set on entry into *Q*. Changes to the floating–point status occurring in *Q* are undone when *Q* returns, with the exception that flags set in *Q* remain set after *Q* returns:

```
MAIN: FLAGS  T F F F F
   Q: FLAGS ON ENTRY:  T F F F F
   Q: ROUNDING MODE ON EXIT: TO_ZERO
   Q: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS  T F F T F
 MAIN: ROUNDING MODE: IEEE_NEAREST
```

When the **–qnostrictieeemod** option is in effect, exception flags set before calling *Q* remain set on entry into *Q*. Changes to the floating-point status occurring in *Q* are propagated to the caller.

```
MAIN: FLAGS  T F F F F
   Q: FLAGS ON ENTRY:  T F F F F
   Q: ROUNDING MODE ON EXIT: TO_ZERO
   Q: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS  F F F T F
```

Chapter 21. Service and utility procedures (IBM extension)

XL Fortran provides utility services that are available to the Fortran programmer. This section describes the rules for the general service and utility procedures, then provides an alphabetical reference to these procedures.

General service and utility procedures

The general service and utility procedures belong to the `xlfutility` module. To ensure that the functions are given the correct type and that naming conflicts are avoided, use these procedures in one of the following two ways:

1. XL Fortran supplies the XLFUTILITY module, which contains the interfaces and data type definitions for these procedures (and the derived-type definitions required for the **dtime_**, **etime_**, **idate_**, and **itime_** procedures). XL Fortran flags arguments that are not compatible with the interface specification in type, kind, and rank. These modules enable type checking of these procedures at compile time rather than at link time. The argument names in the module interface are taken from the examples defined below. The following files are supplied for the `xlfutility` and `xlfutility_extname` modules:

File names	File type	Locations
• xlfutility.f • xlfutility_extname.f	source file	• /opt/ibm/xlf/15.1.0/samples/ modules
• xlfutility.mod • xlfutility_extname.mod	module symbol file	• /opt/ibm/xlf/15.1.0/include

You can use the precompiled module by adding a **USE** statement to your source file (see "USE" on page 483 for details). As well, you can modify the module source file and recompile it to suit your needs. Use the `xlfutility_extname` files for procedures compiled with the **–qextname** option. The source file `xlfutility_extname.f` has no underscores following procedure names, while `xlfutility.f` includes underscores for some procedures names (as listed in this section).

If there are name conflicts (for example if the accessing subprogram has an entity with the same name as a module entity), use the **ONLY** clause or the renaming features of the **USE** statement. For example,

```
USE XLFUTILITY, NULL1 => DTIME_, NULL2 => ETIME_
```

2. Because these procedures are not intrinsic procedures:
   * You must declare their type to avoid potential problems with implicit typing.
   * When compiling with the **–U** option, you must code the names of these procedures in all lowercase to match the names in the XL Fortran libraries. We will show the names in lowercase here as a reminder.

To avoid conflicts with names in the **libc** library, some procedure names end with an underscore. When coding calls to these procedures, you can:

   * Instead of typing the underscore, use the **–qextname** compiler option to add it to the end of each name:
     ```
     xlf –qextname calls_flush.f
     ```

This method is recommended for programs already written without the underscore following the routine name. The XL Fortran library contains additional entry points, such as **fpgets_**, so that calls to procedures that do not use trailing underscores still resolve with **-qextname**.

- Depending on the way your program is structured and the particular libraries and object files it uses, you may have difficulty using **-qextname** or **-brename**. In this case, enter the underscores after the appropriate names in the source file:

```
PRINT *, IRTC()   ! No underscore in this name
CALL FLUSH_(10)   ! But there is one in this name
```

If your program calls the following procedures, there are restrictions on the common block and external procedure names that you can use:

XLF-Provided Function Name	Common Block or External Procedure Name You Cannot Use
mclock	times
rand	irand

List of service and utility procedures

This section lists the service and utility procedures available in the XLFUTILITY module.

Any application that uses the interfaces for the procedures **ctime_**, **gmtime_**, **ltime_**, or **time_** uses the symbolic constant TIME_SIZE to specify the kind type parameter of certain intrinsic data types. The XLFUTILITY module defines TIME_SIZE.

TIME_SIZE is set to 4 for all applications.

Note: CHARACTER(n) means that you can specify any length for the variable.

alarm_(time, func)
Purpose

The **alarm_** function sends an alarm signal **(SIGALRM)** after *time* seconds to invoke the specified function, *func*. This function calls the operating system's alarm system routine.

Class

Function

Argument type and attributes

time   INTEGER(4), INTENT(IN)

func   A function that returns a result of type INTEGER(4).

Result type and attributes

INTEGER(4)

Result value

If a previous alarm request was made with time remaining, **alarm_** returns the remaining time for the previous request in seconds. Otherwise, **alarm_** returns 0.

Examples

```
use, intrinsic :: xlfutility
integer result
integer on_alarm
result = alarm_(100, on_alarm)        ! call on_alarm in 100 seconds
print *, result                       ! prints 0
call sleep_(3)                        ! sleep for 3 seconds
result = alarm_(10, on_alarm)         ! Cancel first alarm. Call on_alarm in 10 seconds
print *, result                       ! prints 97
end

integer function on_alarm()
  on_alarm = 0
end function
```

bic_(X1, X2)
Purpose

The **bic_** subroutine sets bit *X1* of *X2* to 0. For greater portability, it is recommended that you use the IBCLR standard intrinsic procedure instead of this procedure.

Class

Subroutine

Argument type and attributes

X1   INTEGER(4), INTENT(IN)

The range of **X1** must be within 0 to 31, inclusive.

X2   INTEGER(4), INTENT(INOUT)

bis_(X1, X2)
Purpose

The **bis_** subroutine sets bit *X1* of *X2* to 1. For greater portability, it is recommended that you use the IBSET standard intrinsic procedure instead of this procedure.

Class

Subroutine

Argument type and attributes

X1   INTEGER(4), INTENT(IN)

The range of **X1** must be within 0 to 31, inclusive.

X2   INTEGER(4), INTENT(INOUT)

bit_(X1, X2)

Purpose

The **bit_** function returns the value **.TRUE.** if bit *X1* of *X2* equals 1. Otherwise, **bit_** returns the value **.FALSE.**. For greater portability, it is recommended that you use the BTEST standard intrinsic procedure instead of this procedure.

Class

Function

Argument type and attributes

X1   INTEGER(4), INTENT(IN)

The range of **X1** must be within 0 to 31, inclusive.

X2   INTEGER(4), INTENT(IN)

Result type and attributes

LOGICAL(4)

Result value

This function returns .TRUE. if bit **X1** of **X2** equals 1. Otherwise this function returns .FALSE..

clock_()

Purpose

The **clock_** function returns the time in hh:mm:ss format. This function is different from the operating system clock function.

Class

Function

Result type and attributes

CHARACTER(8)

Result value

The time in hh:mm:ss format.

ctime_(STR, TIME)

Purpose

The **ctime_** subroutine converts the system time TIME to a 26-character ASCII string and outputs the result into the first argument. This subroutine calls the operating system's `ctime_r` system routine.

Class

Subroutine

Argument type and attributes

STR   CHARACTER(26), INTENT(OUT)

TIME   INTEGER(KIND=TIME_SIZE), INTENT(IN)

date()
Purpose

The date function returns the current date in mm/dd/yy format.

Class

Function

Result type and attributes

CHARACTER(8)

Result value

The current date in mm/dd/yy format.

dtime_(dtime_struct)
Purpose

The **dtime_** function sets the time accounting information for the user time and system time in DTIME_STRUCT. The resolution for all timing is 1/100 of a second. The output appears in units of seconds.

Class

Function

Argument type and attributes

dtime_struct

```
TYPE TB_TYPE
  SEQUENCE
     REAL(4) USRTIME
     REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) DTIME_STRUCT
```

Result type and attributes

REAL(4)

Result value

The returned value is the sum of the user time and the system time since the last call to **dtime_**.

etime_(etime_struct)

Purpose

The **etime_** function sets the user-elapsed time and system-elapsed time in ETIME_STRUCT since the start of the execution of a process. The resolution for all timing is 1/100 of a second. The output appears in units of seconds.

Class

Function

Argument type and attributes

etime_struct

```
TYPE TB_TYPE
   SEQUENCE
      REAL(4) USRTIME
      REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) ETIME_STRUCT
```

Result type and attributes

REAL(4)

Result value

The returned value is the sum of the user-elapsed time and the system-elapsed time.

exit_(exit_status)

Purpose

The **exit_** subroutine stops execution of the process with exit status *exit_status*. This subroutine calls the operating system's `exit` system routine.

Class

Subroutine

Argument type and attributes

exit_status
INTEGER(4)

fdate_(str)

Purpose

The **fdate_** subroutine returns the date and time in a 26-character ASCII string. The ASCII string is returned in argument STR.

Class

Subroutine

Argument type and attributes

str   CHARACTER(26)

fiosetup_(unit, command, argument)
Purpose

The **fiosetup_** function sets up the requested I/O behavior for the logical unit specified by UNIT. The request is specified by argument COMMAND. The argument ARGUMENT is an argument to the COMMAND. The Fortran include file 'fiosetup_.h' is supplied with the compiler to define symbolic constants for the fiosetup_ arguments and error return codes.

Class

Function

Argument type and attributes

unit   A logical unit that is currently connected to a file

INTEGER(4).

command
INTEGER(4).

IO_CMD_FLUSH_AFTER_WRITE (1). Specifies whether the buffers of the specified UNIT be flushed after every WRITE statement.

IO_CMD_FLUSH_BEFORE_READ (2). Specifies whether the buffers of the specified UNIT be flushed before every READ statement. This can be used to refresh the data currently in the buffers.

argument
INTEGER(4).

IO_ARG_FLUSH_YES (1). Causes the buffers of the specified UNIT to be flushed after every WRITE statement. This argument should be specified with the commands IO_CMD_FLUSH_AFTER_WRITE and IO_CMD_FLUSH_BEFORE_READ.

IO_ARG_FLUSH_NO (0) Instructs the I/O library to flush buffers at its own discretion. Note the units connected to certain device types must be flushed after each WRITE operation regardless of the IO_CMD_FLUSH_AFTER_WRITE setting. Such devices include terminals and pipes. This argument should be specified with the commands IO_CMD_FLUSH_AFTER_WRITE and IO_CMD_FLUSH_BEFORE_READ. This is the default setting for both commands.

Result type and attributes

INTEGER(4).

Result value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors:

IO_ERR_NO_RTE (1000)
The run-time environment is not running.

IO_ERR_BAD_UNIT (1001)
> The specified UNIT is not connected.

IO_ERR_BAD_CMD (1002)
> Invalid command.

IO_ERR_BAD_ARG (1003)
> Invalid argument.

flush_(lunit)
Purpose

The **flush_** subroutine flushes the contents of the input/output buffer for the logical unit LUNIT. The value of LUNIT must be within the range $0 \leq LUNIT \leq 2^{**}31-1$.

For greater portability, use the **FLUSH** statement instead of this procedure.

Class

Subroutine

Argument type and attributes

lunit   INTEGER(4), INTENT(IN)

ftell_(lunit)
Purpose

The **ftell_** function returns the offset of the current byte relative to the beginning of the file associated with the specified logical unit UNIT.

The offset returned by the **ftell_** function is the result of previously completed I/O operations. No references to **ftell_** on a unit with outstanding asynchronous data transfer operations are allowed until the matching **WAIT** statements for all outstanding asynchronous data transfer operations on the same unit are executed.

Class

Function

Argument type and attributes

lunit   INTEGER(4), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

The offset returned by the **ftell_** function is the absolute offset of the current byte relative to the beginning of the file. This means that all bytes from the beginning of the file to the current byte are counted, including the data of the records and record terminators if they are present.

If the unit is not connected, the **ftell_** function returns -1.

ftell64_(lunit)
Purpose

The **ftell64_** function returns the offset of the current byte relative to the beginning of the file associated with the specified logical unit UNIT. The **ftell64** function allows you to query files larger than 2 gigabytes in large file enabled file systems.

The offset returned by the **ftell_** function is the result of previously completed I/O operations. No references to **ftell64_** on a unit with outstanding asynchronous data transfer operations are allowed until the matching **WAIT** statements for all outstanding asynchronous data transfer operations on the same unit are executed.

Class

Function

Argument type and attributes

lunit   INTEGER(4), INTENT(IN)

Result type and attributes

The offset returned by the **ftell64_** function is the absolute offset of the current byte relative to the beginning of the file. This means that all bytes from the beginning of the file to the current byte are counted, including the data of the records and record terminators if they are present.

ftell64_ returns INTEGER(8).

Result value

If the unit is not connected, the **ftell64_** function returns -1.

getarg(i1,c1)
Purpose

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.

For greater portability, use the **GET_COMMAND_ARGUMENT** intrinsic instead of this procedure.

Class

Subroutine

Argument type and attributes

i1   INTEGER(4), INTENT(IN)

c1   CHARACTER(*X*), INTENT(OUT)

　　　　X is the maximum number of characters *c1* can hold.

getcwd_(name)
Purpose

The **getcwd_** function retrieves the pathname NAME of the current working directory where the maximum length is 1024 characters. This function calls the operating system's getcwd system routine.

Class

Function

Argument type and attributes

name   A character string of maximum length 1024

Result type and attributes

INTEGER(4)

Result value

On successful completion, this function returns 0. Otherwise, it returns a system error code (*errno*).

getfd(lunit)
Purpose

Given a Fortran logical unit, the **getfd** function returns the underlying file descriptor for that unit, or -1 if the unit is not connected.

Note: Because XL Fortran does its own I/O buffering, using this function may require special care, as described in Mixed-language input and output in the *XL Fortran Optimization and Programming Guide*.

Class

Function

Argument type and attributes

lunit   INTEGER(4), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

This function returns the underlying file descriptor of the given logical unit, or –1 if the unit is not connected.

getgid_()
Purpose

The **getgid_** function returns the group id of a process, where GROUP_ID is the requested real group id of the calling process. This function calls the operating system's `getgid` system routine.

Class

Function

Result type and attributes

INTEGER(4)

Result value

The group id of a process.

getlog_(name)
Purpose

The **getlog_** subroutine stores the user's login name in NAME. NAME has a maximum length of 8 characters. If the user's login name is not found, NAME is filled with blanks. This subroutine calls the operating system's `getlogin_r` system routine.

Class

Subroutine

Argument type and attributes

name   CHARACTER(8), INTENT(OUT)

getpid_()
Purpose

The **getpid_** function returns the process id of the current process. This function calls the operating system's `getpid` system routine.

Class

Function

Result type and attributes

INTEGER(4)

Result value

The process id of the current process.

getuid_()

Purpose

The **getuid_** function returns the real user id of the current process. This function calls the operating system's `getuid` system routine.

Class

Function

Result type and attributes

INTEGER(4)

Result value

The real user id of the current process.

global_timef()

Purpose

The **global_timef** function returns the elapsed time since the first call to **global_timef** was first executed among all running threads. For thread-specific timing results, see the timef_delta function.

Class

Function

Result type and attributes

REAL(8)

Result value

This function returns in milliseconds, the global timing results from all running threads. The first call to **global_timef** returns 0.0. The accuracy of an XL Fortran timing function is operating system dependent.

gmtime_(stime, tarray)

Purpose

The **gmtime_** subroutine converts the system time STIME into the array TARRAY. The data is stored in TARRAY in the following order:

```
seconds (0 to 59)
minutes (0 to 59)
hours (0 to 23)
day of the month (1 to 31)
month of the year (0 to 11)
year (year = current year - 1900)
day of week (Sunday = 0)
day of year (0 to 365)
daylight saving time (0 or 1)
```

Class

Subroutine

Argument type and attributes

stime   INTEGER(KIND=TIME_SIZE), INTENT(IN)

tarray   INTEGER(4), INTENT(OUT) :: tarray(9)

hostnm_(name)
Purpose

The **hostnm_** function sets *name* to the machine's host name. This function calls the operating system's `gethostname` system routine.

For greater portability, use the **GET_ENVIRONMENT_VARIABLE** intrinsic instead of this procedure.

Class

Function

Argument type and attributes

name   CHARACTER(*X*), INTENT(OUT)

> *X* can be in the range of 1 to 63.

Result type and attributes

INTEGER(4).

Result value

The returned value is 0 if the host name is found, and -1 otherwise.

iargc()
Purpose

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.

For greater portability, use the **COMMAND_ARGUMENT_COUNT** intrinsic instead of this procedure.

Class

Function

Result type and attributes

INTEGER(4)

Result value

The number of arguments.

idate_(idate_struct)
Purpose

The **idate_** subroutine returns the current date in a numerical format containing the day, month and year.

Class

Subroutine

Argument type and attributes

idate_struct

```
TYPE IDATE_TYPE
   SEQUENCE
      INTEGER(4) IDAY
      INTEGER(4) IMONTH
      INTEGER(4) IYEAR
END TYPE
TYPE (IDATE_TYPE) IDATE_STRUCT
```

ierrno_()
Purpose

The **ierrno_** function returns the error number (errno) of the last detected system error.

Class

Function

Result type and attributes

INTEGER(4)

Result value

The error number of the last detected system error.

irand()
Purpose

The **irand** function generates a positive integer number greater than 0 and less than or equal to 32768. The intrinsic subroutine "SRAND(SEED) (IBM extension)" on page 710 is used to provide the seed value for the random number generator.

Class

Function

Result type and attributes

INTEGER(4)

Result value

A pseudo-random positive integer greater than 0 and less than or equal to 32768.

irtc()
Purpose

The **irtc** function returns the number of nanoseconds since the initial value of the machine's real-time clock.

Class

Function

Result type and attributes

INTEGER(8)

Result value

The number of nanoseconds since the initial value of the machine's real-time clock.

itime_(itime_struct)
Purpose

The **itime_** subroutine returns the current time in a numerical form containing seconds, minutes, and hours in ITIME_STRUCT.

Class

Subroutine

Argument type and attributes

itime_struct
```
TYPE IAR
   SEQUENCE
      INTEGER(4) IHR
      INTEGER(4) IMIN
      INTEGER(4) ISEC
END TYPE
TYPE (IAR) ITIME_STRUCT
```

jdate()
Purpose

The **jdate** function returns the current Julian date in yyddd format.

Class

Function

Result type and attributes

CHARACTER(8)

Result value

The current Julian date in yyddd format.

lenchr_(str)
Purpose

The **lenchr_** function returns the length of the given character string.

Class

Function

Argument type and attributes

str   CHARACTER(*), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

The length of the character string.

lnblnk_(str)
Purpose

The **lnblnk_** function returns the index of the last non-blank character in the string STR. If the string contains no non-blank characters, 0 is returned.

Class

Function

Argument type and attributes

str   CHARACTER(*), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

The index of the last non-blank character in the string, or 0 if there are no non-blank characters.

ltime_(stime, tarray)
Purpose

The **ltime_** subroutine dissects the system time STIME, which is in seconds, into the array TARRAY containing the GMT where the dissected time is corrected for the local time zone. The data is stored in TARRAY in the following order:

```
seconds (0 to 59)
minutes (0 to 59)
hours (0 to 23)
day of the month (1 to 31)
month of the year (0 to 11)
year (year = current year - 1900)
day of week (Sunday = 0)
day of year (0 to 365)
daylight saving time (0 or 1)
```

Class

Subroutine

Argument type and attributes

stime INTEGER(KIND=TIME_SIZE), INTENT(IN)

tarray INTEGER(4), INTENT(OUT):: tarry(9)

mclock()
Purpose

The **mclock** function returns time accounting information about the current process and its child processes. The accuracy of an XL Fortran timing function is operating system dependent.

Class

Function

Result type and attributes

INTEGER(4)

Result value

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 one-hundredth (1/100) of a second.

qsort_(array, len, isize, compar)
Purpose

The **qsort_** subroutine performs a parallel quicksort on a one-dimensional array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE, and a user-defined sorting order function COMPAR to sort the elements of the array.

Class

Subroutine

Argument type and attributes

array   The array to be sorted. It can be of any type.

len   The number of elements in the array. The argument is of type INTEGER(4).

isize   The size of a single element of the array. The argument is of type INTEGER(4).

compar
> A user-defined comparison function used to sort the array.

Examples

```
INTEGER(4) FUNCTION COMPAR_UP(C1, C2)
INTEGER(4) C1, C2
IF (C1.LT.C2) COMPAR_UP = -1
IF (C1.EQ.C2) COMPAR_UP =  0
IF (C1.GT.C2) COMPAR_UP =  1
RETURN
END

SUBROUTINE FOO()
  INTEGER(4) COMPAR_UP
  EXTERNAL COMPAR_UP
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 6
  ISIZE = 4
  CALL qsort_(ARRAY(3:8), LEN, ISIZE, COMPAR_UP)! sorting ARRAY(3:8)
  PRINT *, ARRAY                 ! result value is [0, 3, 1, 2, 4, 5, 7, 9]
  RETURN
END
```

qsort_down(array, len, isize)
Purpose

The **qsort_down** subroutine performs a parallel quicksort on a one-dimensional array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE. The result is stored in array ARRAY in descending order. As opposed to **qsort_**, the **qsort_down** subroutine does not require the COMPAR function. The **qsort_down** is not suitable for sorting character(N) arrays where N is larger than 1. The **qsort_ routine** can be used to sort such arrays.

Class

Subroutine

Argument type and attributes

array   The array to be sorted. It can be of any type.

len   The number of elements in the array. The argument is of type INTEGER(4).

isize   The size of a single element of the array. The argument is of type INTEGER(4).

Examples

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
  CALL qsort_down(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [9, 7, 5, 4, 3, 2, 1, 0]
RETURN
END
```

qsort_up(array, len, isize)

Purpose

The **qsort_up** subroutine performs a parallel quicksort on a one-dimensional, contiguous array ARRAY whose length LEN is the number of elements in the array with each element having a size of ISIZE. The result is stored in array ARRAY in ascending order. As opposed to **qsort_**, the **qsort_up** subroutine does not require the COMPAR function. The **qsort_up** is not suitable for sorting character(N) arrays where N is larger than 1. The **qsort_ routine** can be used to sort such arrays.

Class

Subroutine

Argument type and attributes

array   The array to be sorted. It can be of any type.

len   The number of elements in the array. The argument is of type INTEGER(4).

isize   The size of a single element of the array. The argument is of type INTEGER(4).

Examples

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
  CALL qsort_up(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [0, 1, 2, 3, 4, 5, 7, 9]
RETURN
END
```

rtc()

Purpose

The **rtc** function returns the number of seconds since the initial value of the machine's real-time clock.

Class

Function

Result type and attributes

REAL(8)

Result value

The number of seconds since the initial value of the machine's real-time clock.

setrteopts(c1)
Purpose

The **setrteopts** subroutine changes the setting of one or more of the run-time options during the execution of a program. See *Setting Run-Time Options* in the *XL Fortran Compiler Reference* for details about the run-time options.

Class

Subroutine

Argument type and attributes

c1   CHARACTER(*X*), INTENT(IN)

X is the length of the run-time option to be set.

sleep_(sec)
Purpose

The **sleep_** subroutine suspends the execution of the current process for *sec* seconds.

Class

Subroutine

Argument type and attributes

sec   INTEGER(4), INTENT(IN)

time_()
Purpose

The **time_** function returns the current time (GMT), in seconds, since the Epoch. This function calls the operating system's `time` system routine.

Class

Function

Result type and attributes

INTEGER(KIND=TIME_SIZE).

Result value

The current time (GMT), in seconds.

timef()

Purpose

The **timef** function returns the elapsed time in milliseconds since the first call to **timef**. The accuracy of an XL Fortran timing function is operating system dependent.

Class

Function

Result type and attributes

REAL(8)

Result value

The elapsed time in milliseconds since the first call to timef. The first call to **timef** returns 0.0d0.

timef_delta(t)

Purpose

The **timef_delta** function returns the elapsed time in milliseconds since the last instance **timef_delta** was called with its argument set to 0.0 within the same thread. In order to get the correct elapsed time, you must determine which region of a thread you want timed. This region must start with a call to **timef_delta(T0)**, where T0 is initialized (T0=0.0). The next call to **timef_delta** must use the first call's return value as the input argument if the elapsed time is expected. The accuracy of an XL Fortran timing function is operating system dependent.

Class

Function

Argument type and attributes

t   REAL(8)

Result type and attributes

REAL(8)

Result value

Time elapsed in milliseconds.

umask_(cmask)

Purpose

The **umask_** function sets the file mode creation mask to CMASK. This function calls the operating system's umask system routine.

Class

Function

Argument type and attributes

cmask INTEGER(4), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

The returned value is the previous value of the file mode creation mask.

usleep_(msec)
Purpose

The **usleep_** function suspends the execution of the current process for an interval of MSEC microseconds. This function calls the operating system's usleep system routine. The accuracy of the result is, therefore, operating system dependent.

Class

Function

Argument type and attributes

msec   INTEGER(4), INTENT(IN)

Result type and attributes

INTEGER(4)

Result value

The returned value is 0 if the function is successful, or an error number otherwise.

xl__trbk()
Purpose

The **xl__trbk** subroutine provides a traceback starting from the invocation point. **xl__trbk** can be called from your code, although not from signal handlers. The subroutine requires no parameters.

Class

Subroutine

The XLF_POSIX_BINDINGS module

The XLF_POSIX_BINDINGS module provides interfaces to many POSIX and XSI functions and macros. You can consult your operating system documentation or the Open Group Base Specifications for the exact meaning and semantics of these functions and macros.

The XLF_POSIX_BINDINGS module behaves as if it contained a **USE** statement for the ISO_C_BINDING intrinsic module. All values that are public in the ISO_C_BINDING module remain public in the XLF_POSIX_BINDINGS module.

Types

The XLF_POSIX_BINDINGS module contains the definition of type `utsname`. Derived type `utsname` contains the following character components:

machine
> Machine ID number.

nodename
> Host name on the network.

release
> Operating system release number.

sysname
> Operating system name.

version
> Operating system version number.

Named constants

The XLF_POSIX_BINDINGS module contains the definitions of the following named constants.

Named constants as integer kinds

The following named constants can be used as integer kinds:

Table 98. Correspondence between XLF_POSIX_BINDINGS named constants and C type definition

Named constant	Value	Corresponding C type definition
C_CLOCK_T	8	clock_t
C_GID_T	4	gid_t
C_MODE_T	4	mode_t
C_OFF_T	8	off_t
C_PID_T	4	pid_t
C_TIME_T	8	time_t
C_UID_T	4	uid_t
C_USECONDS_T	4	useconds_t

Named constants corresponding to POSIX and XSI macros

The following named constants correspond to POSIX and XSI macros. The values of the named constants correspond to a _POSIX_C_SOURCE level of `200809L` and an _XOPEN_SOURCE level of `700`.

The named constants fall into the following categories:
- Based on `errno.h`
  - Named constants corresponding to `errno` values
- Based on `fcntl.h`

- Named constants used with the `chmod`, `mkdir`, `mkfifo`, and `umask` functions
- Named constants based on `fcntl.h`
• Based on `stdlib.h`
  - Named constants used with the `exit` function
• Based on `unistd.h`
  - Named constants used with the `sysconf` function
  - Named constants used with the `access` function
  - Named constants used with the `pathconf` function
  - Other constants
• Based on `signal.h`
  - Named constants corresponding to signal names
  - Other constants
• Based on `time.h`
• Based on `limits.h`
• Based on `sys/wait.h`

Named constants that are based on `errno.h`:

Table 99. Named constants corresponding to values returned by the `errno` function:

E2BIG	EFAULT	ENOBUFS	ENXIO
EACCES	EFBIG	ENODATA	EOPNOTSUPP
EADDRINUSE	EIDRM	ENODEV	EOVERFLOW
EADDRNOTAVAIL	EILSEQ	ENOENT	EPERM
EAFNOSUPPORT	EINPROGRESS	ENOEXEC	EPIPE
EAGAIN	EINTR	ENOLCK	EPROTONOSUPPORT
EALREADY	EINVAL	ENOLINK	EPROTOTYPE
EBADF	EIO	ENOMEM	EPROTO
EBADMSG	EISCONN	ENOMSG	ERANGE
EBUSY	EISDIR	ENOPROTOOPT	EROFS
ECANCELED	ELOOP	ENOSPC	ESPIPE
ECHILD	EMFILE	ENOSR	ESRCH
ECONNABORTED	EMLINK	ENOSTR	ESTALE
ECONNREFUSED	EMSGSIZE	ENOSYS	ETIMEDOUT
ECONNRESET	EMULTIHOP	ENOTCONN	ETIME
EDEADLK	ENAMETOOLONG	ENOTDIR	ETXTBSY
EDESTADDRREQ	ENETDOWN	ENOTEMPTY	EWOULDBLOCK
EDOM	ENETRESET	ENOTSOCK	EXDEV
EDQUOT	ENETUNREACH	ENOTSUP	
EEXIST	ENFILE	ENOTTY	

Named constants that are based on `fcntl.h`:

Table 100. Named constants that are used with the `chmod`, `mkdir`, `mkfifo`, and `umask` functions

S_IRWXU	S_IWGRP
S_IRUSR	S_IXGRP
S_IWUSR	S_IRWXO
S_IXUSR	S_IROTH
S_IRWXG	S_IWOTH
S_IRGRP	S_IXOTH

Table 101. Named constants based on fcntl.h

F_DUPFD	F_RDLCK	O_SYNC	S_IFMT
F_GETFD	F_UNLCK	O_ACCMODE	S_IFBLK
F_SETFD	F_WRLCK	O_RDONLY	S_IFCHR
F_GETFL	O_CREAT	O_RDWR	S_IFIFO
F_SETFL	O_EXCL	O_WRONLY	S_IFREG
F_GETLK	O_NOCTTY	POSIX_FADV_NORMAL	S_IFDIR
F_SETLK	O_TRUNC	POSIX_FADV_SEQUENTIAL	S_IFLNK
F_SETLKW	O_APPEND	POSIX_FADV_RANDOM	S_IFSOCK
F_GETOWN	O_DSYNC	POSIX_FADV_WILLNEED	S_ISUID
F_SETOWN	O_NONBLOCK	POSIX_FADV_DONTNEED	S_ISGID
FD_CLOEXEC	O_RSYNC	POSIX_FADV_NOREUSE	S_ISVTX

Named constants that are based on stdlib.h:

Table 102. Named constants used with the exit function

EXIT_FAILURE	EXIT_SUCCESS

Named constants that are based on unistd.h:

Table 103. Named constants used with the `sysconf` function

_SC_2_CHAR_TERM	_SC_RTSIG_MAX
_SC_2_C_BIND	_SC_SAVED_IDS
_SC_2_C_DEV	_SC_SEMAPHORES
_SC_2_FORT_DEV	_SC_SEM_NSEMS_MAX
_SC_2_FORT_RUN	_SC_SEM_VALUE_MAX
_SC_2_LOCALEDEF	_SC_SHARED_MEMORY_OBJECTS
_SC_2_PBS	_SC_SHELL
_SC_2_PBS_ACCOUNTING	_SC_SIGQUEUE_MAX
_SC_2_PBS_CHECKPOINT	_SC_SPAWN
_SC_2_PBS_LOCATE	_SC_SPIN_LOCKS
_SC_2_PBS_MESSAGE	_SC_SPORADIC_SERVER
_SC_2_PBS_TRACK	_SC_SS_REPL_MAX
_SC_2_SW_DEV	_SC_STREAM_MAX
_SC_2_UPE	_SC_SYMLOOP_MAX
_SC_2_VERSION	_SC_SYNCHRONIZED_IO
_SC_ADVISORY_INFO	_SC_THREADS
_SC_AIO_LISTIO_MAX	_SC_THREAD_ATTR_STACKADDR
_SC_AIO_MAX	_SC_THREAD_ATTR_STACKSIZE
_SC_AIO_PRIO_DELTA_MAX	_SC_THREAD_CPUTIME
_SC_ARG_MAX	_SC_THREAD_DESTRUCTOR_ITERATIONS
_SC_ASYNCHRONOUS_IO	_SC_THREAD_KEYS_MAX
_SC_ATEXIT_MAX	_SC_THREAD_PRIORITY_SCHEDULING
_SC_BARRIERS	_SC_THREAD_PRIO_INHERIT
_SC_BC_BASE_MAX	_SC_THREAD_PRIO_PROTECT
_SC_BC_DIM_MAX	_SC_THREAD_PROCESS_SHARED
_SC_BC_SCALE_MAX	_SC_THREAD_SAFE_FUNCTIONS
_SC_BC_STRING_MAX	_SC_THREAD_SPORADIC_SERVER
_SC_CHILD_MAX	_SC_THREAD_STACK_MIN
_SC_CLK_TCK	_SC_THREAD_THREADS_MAX
_SC_CLOCK_SELECTION	_SC_TIMEOUTS
_SC_COLL_WEIGHTS_MAX	_SC_TIMERS
_SC_CPUTIME	_SC_TIMER_MAX
_SC_DELAYTIMER_MAX	_SC_TRACE
_SC_EXPR_NEST_MAX	_SC_TRACE_EVENT_FILTER
_SC_FSYNC	_SC_TRACE_EVENT_NAME_MAX
_SC_GETGR_R_SIZE_MAX	_SC_TRACE_INHERIT
_SC_GETPW_R_SIZE_MAX	_SC_TRACE_LOG
_SC_HOST_NAME_MAX	_SC_TRACE_NAME_MAX
_SC_IOV_MAX	_SC_TRACE_SYS_MAX
_SC_IPV6	_SC_TRACE_USER_EVENT_MAX
_SC_JOB_CONTROL	_SC_TTY_NAME_MAX
_SC_LINE_MAX	_SC_TYPED_MEMORY_OBJECTS
_SC_LOGIN_NAME_MAX	_SC_TZNAME_MAX
_SC_MAPPED_FILES	_SC_V6_ILP32_OFF32
_SC_MEMLOCK	_SC_V6_ILP32_OFFBIG
_SC_MEMLOCK_RANGE	_SC_V6_LP64_OFF64
_SC_MEMORY_PROTECTION	_SC_V6_LPBIG_OFFBIG
_SC_MESSAGE_PASSING	_SC_VERSION
_SC_MONOTONIC_CLOCK	_SC_XBS5_ILP32_OFF32
_SC_MQ_OPEN_MAX	_SC_XBS5_ILP32_OFFBIG
_SC_MQ_PRIO_MAX	_SC_XBS5_LP64_OFF64
_SC_NGROUPS_MAX	_SC_XBS5_LPBIG_OFFBIG
_SC_OPEN_MAX	_SC_XOPEN_CRYPT
_SC_PAGESIZE	_SC_XOPEN_ENH_I18N
_SC_PAGE_SIZE	_SC_XOPEN_LEGACY
_SC_PRIORITIZED_IO	_SC_XOPEN_REALTIME
_SC_PRIORITY_SCHEDULING	_SC_XOPEN_REALTIME_THREADS
_SC_RAW_SOCKETS	_SC_XOPEN_SHM
_SC_READER_WRITER_LOCKS	_SC_XOPEN_STREAMS
_SC_REALTIME_SIGNALS	_SC_XOPEN_UNIX
_SC_REGEXP	_SC_XOPEN_VERSION
_SC_RE_DUP_MAX	

Table 104. Named constants used with the `access` function

F_OK	R_OK	W_OK	X_OK

Table 105. Named constants used with the `pathconf` *function*

_PC_CHOWN_RESTRICTED	_PC_SYNC_IO
_PC_LINK_MAX	_PC_PRIO_IO
_PC_MAX_CANON	_PC_FILESIZEBITS
_PC_MAX_INPUT	_PC_SYMLINK_MAX
_PC_NAME_MAX	_PC_ALLOC_SIZE_MIN
_PC_NO_TRUNC	_PC_REC_INCR_XFER_SIZE
_PC_PATH_MAX	_PC_REC_MAX_XFER_SIZE
_PC_PIPE_BUF	_PC_REC_MIN_XFER_SIZE
_PC_VDISABLE	_PC_REC_XFER_ALIGN
_PC_ASYNC_IO	_PC_2_SYMLINKS

Table 106. Other constants

_POSIX_VERSION	_POSIX_THREAD_SPORADIC_SERVER
_POSIX2_VERSION	_POSIX_THREADS
_XOPEN_VERSION	_POSIX_TIMEOUTS
_POSIX_ADVISORY_INFO	_POSIX_TIMERS
_POSIX_ASYNCHRONOUS_IO	_POSIX_TRACE
_POSIX_BARRIERS	_POSIX_TRACE_EVENT_FILTER
_POSIX_CHOWN_RESTRICTED	_POSIX_TRACE_INHERIT
_POSIX_CLOCK_SELECTION	_POSIX_TRACE_LOG
_POSIX_CPUTIME	_POSIX_TYPED_MEMORY_OBJECTS
_POSIX_FSYNC	_POSIX_VDISABLE
_POSIX_IPV6	_POSIX2_CHAR_TERM
_POSIX_JOB_CONTROL	_POSIX2_C_BIND
_POSIX_MAPPED_FILES	_POSIX2_C_DEV
_POSIX_MEMLOCK	_POSIX2_LOCALEDEF
_POSIX_MEMLOCK_RANGE	_POSIX2_SW_DEV
_POSIX_MEMORY_PROTECTION	_XOPEN_CRYPT
_POSIX_MESSAGE_PASSING	_XOPEN_ENH_I18N
_POSIX_MONOTONIC_CLOCK	_XOPEN_LEGACY
_POSIX_NO_TRUNC	_XOPEN_REALTIME
_POSIX_PRIORITIZED_IO	_XOPEN_REALTIME_THREADS
_POSIX_PRIORITY_SCHEDULING	_XOPEN_SHM
_POSIX_RAW_SOCKETS	_CS_PATH
_POSIX_READER_WRITER_LOCKS	_XOPEN_UNIX
_POSIX_REALTIME_SIGNALS	SEEK_CUR
_POSIX_REGEXP	SEEK_END
_POSIX_SAVED_IDS	SEEK_SET
_POSIX_SEMAPHORES	
_POSIX_SHARED_MEMORY_OBJECTS	
_POSIX_SHELL	
_POSIX_SPAWN	
_POSIX_SPIN_LOCKS	
_POSIX_SPORADIC_SERVER	
_POSIX_SYNCHRONIZED_IO	
_POSIX_THREAD_ATTR_STACKADDR	
_POSIX_THREAD_ATTR_STACKSIZE	
_POSIX_THREAD_CPUTIME	
_POSIX_THREAD_PRIO_INHERIT	
_POSIX_THREAD_PRIO_PROTECT	
_POSIX_THREAD_PRIORITY_SCHEDULING	
_POSIX_THREAD_PROCESS_SHARED	
_POSIX_THREAD_SAFE_FUNCTIONS	

Named constants that are based on `signal.h`:

Table 107. Named constants corresponding to signal names that can be specified in the `kill`, `killpg`, `raise`, *and* `signal` *functions*

SIGABRT	SIGILL	SIGTERM	SIGPROF
SIGALRM	SIGINT	SIGTSTP	SIGSYS
SIGBUS	SIGKILL	SIGTTIN	SIGTRAP
SIGCHLD	SIGPIPE	SIGTTOU	SIGURG
SIGCONT	SIGQUIT	SIGUSR1	SIGVTALRM
SIGFPE	SIGSEGV	SIGUSR2	SIGXCPU
SIGHUP	SIGSTOP	SIGPOLL	SIGXFSZ

Table 108. Other constants

BUS_ADRALN	FPE_INTDIV	POLL_OUT	SIG_BLOCK
BUS_ADRERR	FPE_INTOVF	POLL_PRI	SIG_SETMASK
BUS_OBJERR	ILL_BADSTK	SA_NOCLDSTOP	SIG_UNBLOCK
CLD_CONTINUED	ILL_COPROC	SA_NOCLDWAIT	SI_ASYNCIO
CLD_DUMPED	ILL_ILLADR	SA_NODEFER	SI_MESGQ
CLD_EXITED	ILL_ILLOPC	SA_ONSTACK	SI_QUEUE
CLD_KILLED	ILL_ILLOPN	SA_RESETHAND	SI_TIMER
CLD_STOPPED	ILL_ILLTRP	SA_RESTART	SI_USER
CLD_TRAPPED	ILL_PRVOPC	SA_SIGINFO	SS_DISABLE
FPE_FLTDIV	ILL_PRVREG	SEGV_ACCERR	SS_ONSTACK
FPE_FLTINV	MINSIGSTKSZ	SEGV_MAPERR	TRAP_BRKPT
FPE_FLTOVF	POLL_ERR	SIGEV_NONE	TRAP_TRACE
FPE_FLTRES	POLL_HUP	SIGEV_SIGNAL	
FPE_FLTSUB	POLL_IN	SIGEV_THREAD	
FPE_FLTUND	POLL_MSG	SIGSTKSZ	

Named constants that are based on `time.h`:

Table 109. Named constants based on `time.h`

CLOCKS_PER_SEC	CLOCK_PROCESS_CPUTIME_ID	CLOCK_THREAD_CPUTIME_ID

Named constants based on `limits.h`:NZERO

Note: You can get the values of other limits by calling the `sysconf` and `pathconf` functions.

Named constants that are based on `sys/wait.h`:

Table 110. Named constants based on `sys/wait.h`

WCONTINUED	WNOHANG	WUNTRACED

Abstract interfaces

The XLF_POSIX_BINDINGS module defines the `compar_iface` abstract interface as follows:

```
ABSTRACT INTERFACE
  FUNCTION compar_iface(a, b)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT
    IMPLICIT NONE
    TYPE(*), INTENT(IN) :: a, b
    INTEGER(C_INT) compar_iface
  END FUNCTION
END INTERFACE
```

Procedures

This section lists the procedures that are available in the XLF_POSIX_BINDINGS module.

access(path, amode)
Purpose

Queries the accessibility of a file.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

amode
> An **INTEGER**(C_INT). The value of the corresponding actual argument must be one or more of the R_OK, W_OK, X_OK, or F_OK named constants.

Result type and attributes

INTEGER(C_INT)

Result value

If the requested access is permitted, the result is 0. Otherwise, the result is -1.

This function sets the error number (errno).

Examples

```
SUBROUTINE sub(filename)
  USE XLF_POSIX_BINDINGS
  CHARACTER(*) filename

  IF (access(filename, F_OK) == 0) THEN
  ! The requested file exists.
    IF (access(filename, ior(R_OK, W_OK)) == -1) THEN
      ! File does not have read and write permission,
      ! or an error occurs. You can use errno() for more information.
      ...
    END IF
  END IF
END SUBROUTINE

...
CHARACTER(20) myfile
myfile = '/home/user/file' // C_NULL_CHAR
CALL sub(myfile)
CALL sub('/home/user/file')
```

alarm(seconds)
Purpose

Requests that the operating system generate an alarm signal (SIGALARM) after **seconds** seconds.

Class

Function

Argument type and attributes

seconds
An **INTEGER**(4). The corresponding actual argument must be larger than or equal to zero. If the corresponding actual argument is 0, any pending alarm request is canceled.

Result type and attributes

INTEGER(4)

Result value

If there is a pending alarm request, the result is the number of seconds remaining until the pending request. Otherwise, the result is 0.

Related information
- "alarm_(time, func)" on page 968
- "ualarm(useconds, interval)" on page 1034

atexit(x)
Purpose

Registers a subroutine to be called at normal program termination.

Class

Function

Argument type and attributes

x A subroutine. The corresponding actual argument cannot have dummy arguments.

Result type and attributes

INTEGER(C_INT)

Result value

If the subroutine was registered successfully, the result is 0. Otherwise, the result is nonzero.

Examples
```
PROGRAM main
  USE XLF_POSIX_BINDINGS
  INTEGER(c_int) result
```

```
      result = atexit(goodbye)
      PRINT *, 'hello'
      CONTAINS
        SUBROUTINE goodbye()
          PRINT *, 'goodbye'
        END SUBROUTINE
END PROGRAM
```

The output is as follows:
```
hello
goodbye
```

bsearch(key, base, nel, width, compar)
Purpose

Binary search of an array.

Class

Function

Argument type and attributes

key
> A scalar of any type.

base
> An **INTENT(INOUT)** contiguous array. The actual argument corresponding to
> **base** must be of the same type and type parameters as the actual argument
> corresponding to **key**.

nel
> An **INTEGER**(C_SIZE_T). The value of the corresponding actual argument must
> equal the number of elements in **base**.

width
> An **INTEGER**(C_SIZE_T). The value of the corresponding actual argument must
> equal the size of each element in **base**.

compar
> A procedure with an interface that conforms to compar_iface.

Result type and attributes

TYPE(C_PTR)

Result value

If the value of **key** is found in array **base**, the result value is a C_PTR to an instance
of the value of **key** in **base**. Otherwise, the result is C_NULL_PTR.

Examples
```
MODULE pts
  USE XLF_POSIX_BINDINGS
  IMPLICIT NONE
  TYPE, BIND(C) :: pt
    INTEGER x, y
  END TYPE
  CONTAINS
    FUNCTION y_less_than(a, b)
```

```
      TYPE(pt), INTENT(IN) :: a, b
      INTEGER(C_INT) y_less_than
      y_less_than = a%y - b%y
    END FUNCTION
END MODULE

USE pts
IMPLICIT NONE
TYPE(pt), TARGET :: a(5)
TYPE(C_PTR) :: result
TYPE(pt), POINTER :: fresult

a = [pt(2,0), pt(7,1), pt(9,2), pt(1,7), pt(5,7)]

! Find a point with y == 7.
result = bsearch(pt(0,7), a, SIZE(a), C_SIZEOF(a(1)), y_less_than)
IF (C_ASSOCIATED(result)) THEN
  CALL c_f_pointer(result, fresult)
  PRINT *, 'found:', fresult
ELSE
  PRINT *, 'found nothing.'
END IF
END
```

The output is as follows:

```
found: 5 7
```

Related information
- "lfind(key, base, nelp, width, compar)" on page 1017
- "lsearch(key, base, nelp, width, compar)" on page 1018

calloc(nelem, elsize)
Purpose

Allocates an array of **nelem** elements, each of **elsize** bytes, and sets all bits in the array to 0.

Class

Function

Argument type and attributes

nelem
    An **INTEGER**(C_SIZE_T).

elsize
    An **INTEGER**(C_SIZE_T).

Result type and attributes

INTEGER(C_INTPTR_T)

Result value

The result is the address of the allocated memory block, or 0 if unsuccessful. The use of the **calloc** procedure is discouraged because it can lead to memory leaks. It is preferred to use allocatable variables instead.

This function sets the error number (errno).

Related information
- "ALLOCATABLE (Fortran 2003)" on page 285
- "free(ptr)" on page 1002
- "malloc(size)" on page 1019
- "realloc(ptr, size)" on page 1024

chdir(path)
Purpose

Changes the working directory.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

chmod(path, mode)
Purpose

Changes access permission to a file.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

mode
> An **INTEGER**(C_MODE_T). The value of the corresponding actual argument must be one or more of the named constants from Table 100 on page 990.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

chown(path, owner, group)
Purpose

Changes the owner and group of a file.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

owner
> An **INTEGER**(C_UID_T).

group
> An **INTEGER**(C_UID_T).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

clock()
Purpose

Returns the CPU time used by the calling process.

Class

Function

Result type and attributes

INTEGER(C_CLOCK_T)

Result value

The result is the CPU time used by the calling process. To convert this value to seconds, you can divide the result by `CLOCKS_PER_SEC`.

If the CPU time used by the process is unavailable, the result is -1.

confstr(name, buf)
Purpose

Gets the value of a configurable system limit or variable.

Class

Function

Argument type and attributes

name
> An **INTEGER**(`C_INT`). The value of the corresponding actual argument must be the name of system constant.

buf
> An **INTENT(OUT) CHARACTER**(*).

Result type and attributes

INTEGER(`C_SIZE_T`)

Result value

If **name** has a configurable value, the result is the length of buffer required to hold the configurable value including null-termination. If the result is larger than `LEN(buf)` + 1, the configurable value stored in **buf** is truncated. If **name** does not have a configurable value, or if there is an error, the result is 0.

This function sets the error number (`errno`).

Related information

errno()
Purpose

Gets the current value of the error number (`errno`).

Class

Function

Result type and attributes

INTEGER(`C_INT`)

Result value

The result is the value of errno.

This function has the same functionality as **ierrno_**.

Related information
• "ierrno_()" on page 980

exit(status)
Purpose

Terminates the calling process with exit status **status**.

Class

Subroutine

Argument type and attributes

status
An **INTEGER**(C_INT). The value of the corresponding actual argument must be
EXIT_SUCCESS, EXIT_FAILURE, or in the range 0 to 255 inclusive. If the value of
the corresponding actual argument is outside this range, only the least
significant 8 bits will be available to any waiting parent process.

fork()
Purpose

Creates a new process.

Class

Function

Result type and attributes

INTEGER(C_PID_T)

Result value

If the new process is created successfully, the result for the old process is the
process ID of the new child process, and the result for the child process is 0. If the
new process could not be created, the result is -1.

This function sets the error number (errno).

free(ptr)
Purpose

Deallocates a block of memory that was allocated by **calloc**, **malloc**, or **realloc**.

Class

Subroutine

Argument type and attributes

ptr

An **INTEGER**(C_INTPTR_T). The corresponding actual argument must have a value obtained from a call to **calloc**, **malloc**, or **realloc**.

Related information
- "calloc(nelem, elsize)" on page 998
- "malloc(size)" on page 1019
- "realloc(ptr, size)" on page 1024

getcwd(buf)
Purpose

Gets the path name of the current working directory.

Class

Function

Argument type and attributes

buf

An **INTENT(OUT) CHARACTER**(*).

Result type and attributes

TYPE(C_PTR)

Result value

If successful, the result is a C_PTR that points to the actual argument corresponding to **buf**. If unsuccessful, the result is C_NULL_PTR.

This function sets the error number (errno).

Examples

```
USE XLF_POSIX_BINDINGS
TYPE(C_PTR) res
INTEGER(C_LONG) path_len
CHARACTER(:), ALLOCATABLE :: path

path_len = pathconf('.', _PC_PATH_MAX);
IF (path_len /= -1) THEN
  ALLOCATE(CHARACTER(path_len) :: path)
  res = getcwd(path)
  IF (C_ASSOCIATED(res)) THEN
    PRINT *, 'current directory is ', trim(path)
  END IF
ELSE
  CALL perror('')
END IF
END
```

The output is as follows:

```
current directory is /home/user
```

Related information
- "getcwd_(name)" on page 976

getegid()
Purpose

Gets the effective group ID.

Class

Pure function

Result type and attributes

INTEGER(C_GID_T)

Result value

The result is the effective group ID of the process.

geteuid()
Purpose

Gets the effective user ID.

Class

Pure function

Result type and attributes

INTEGER(C_UID_T)

Result value

The result is the effective user ID of the process.

getgid()
Purpose

Gets the group ID.

Class

Pure function

Result type and attributes

INTEGER(C_GID_T)

Result value

The result is the real group ID of the process.

getgroups(gidsetsize, grouplist)
Purpose

Gets supplementary group IDs.

Class

Function

Argument type and attributes

gidsetsize
> An **INTEGER**(C_INT). The value of the corresponding actual argument must equal the number of elements in **grouplist**.

grouplist
> An **INTENT(INOUT) INTEGER(C_GID_T)** array. When the value of the actual argument corresponding to **gidsetsize** is not zero, **grouplist** is assigned the supplementary group IDs of the calling process.

Result type and attributes

INTEGER(C_INT)

Result value

If **gidsetsize** is not zero, the result is the number of group IDs assigned to **grouplist**. If **gidsetsize** is zero, the result is the number of supplementary group IDs of the calling process. If an error occurred, the result is -1.

This function sets the error number errno.

gethostid()
Purpose

Gets the host identifier.

Class

Function

Result type and attributes

INTEGER(C_LONG)

Result value

The result is a 32-bit identifier for the current host.

gethostname(name)
Purpose

Gets the host name

Class

Function

Argument type and attributes

name
> An **INTENT(OUT) CHARACTER**(*). **name** is assigned the name of the current host. If **name** is not large enough to hold the host name, the host name is truncated.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

getlogin_r(name)
Purpose

Gets the user name that is used to log into the controlling terminal of the calling process.

Class

Function

Argument type and attributes

name
> An **INTENT(OUT) CHARACTER**(*). **name** is assigned the user name used to log into the controlling terminal of the current process.

Result type and attributes

INTEGER(C_INT)

Result value

If successful, the result is 0. If unsuccessful, the result is a nonzero error number.

This function sets the error number (errno).

getpgid(pid)
Purpose

Gets the group ID of a process

Class

Function

Argument type and attributes

pid
> An **INTEGER**(C_PID_T).

Result type and attributes

INTEGER(C_PID_T)

Result value

If **pid** is nonzero, the result is the group ID associated with the process with ID **pid**. If **pid** is zero, the result is the group ID of the calling process. If unsuccessful, the result is -1.

This function sets the error number (`errno`).

getpgrp()
Purpose

Gets the process group ID of the calling process.

Class

Function

Result type and attributes

INTEGER(`C_PID_T`)

Result value

The result is the process group ID of the calling process.

getpid()
Purpose

Gets the process ID of the calling process.

Class

Pure function

Result type and attributes

INTEGER(`C_PID_T`)

Result value

The result is the process ID of the calling process.

getppid()
Purpose

Gets the parent process ID.

Class

Function

Result type and attributes

INTEGER(`C_PID_T`)

Result value

The result is the parent process ID.

getsid(pid)
Purpose

Gets the process group ID of a session leader.

Class

Function

Argument type and attributes

pid
    An **INTEGER**(C_PID_T).

Result type and attributes

INTEGER(C_PID_T)

Result value

The result is the process group ID of the session leader of the process with ID **pid**. If unsuccessful, the result is -1.

This function sets the error number (errno).

getuid()
Purpose

Gets the real user ID of the calling process.

Class

Pure function

Result type and attributes

INTEGER(C_UID_T)

Result value

The result is the real user ID of the calling process.

isalnum(c)
Purpose

Tests for an alphanumeric character.

Class

Pure function

Argument type and attributes

c An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) is an alphanumeric character in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isalpha(c)
Purpose

Tests for an alphabetic character.

Class

Pure function

Argument type and attributes

c An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) is an alphabetic character in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isascii(c)
Purpose

Tests for a 7-bit US-ASCII character.

Class

Pure function

Argument type and attributes

c An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) is a 7-bit US-ASCII character. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isblank(c)
Purpose

Tests for a blank character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the blank class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

iscntrl(c)
Purpose

Tests for a control character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the `cntrl` character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isdigit(c)
Purpose

Tests for a decimal digit character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the `digit` character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isgraph(c)
Purpose

Tests for a visible character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the `graph` character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

islower(c)
Purpose

Tests for a lowercase character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the `lower` character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isprint(c)
Purpose

Tests for a printable character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the `print` character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

ispunct(c)
Purpose

Tests for a punctuation character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the punct character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isupper(c)
Purpose

Tests for an uppercase character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the upper character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

isxdigit(c)
Purpose

Tests for a hexadecimal digit character.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if **ACHAR**(**c**) belongs to the xdigit character class in the locale of the calling process. Otherwise, the result is 0.

Related information

"ACHAR(I, KIND)" on page 574

j0(x)
Purpose

Bessel function of the first kind, order 0.

Class

Pure function

Argument type and attributes

x   A **REAL**(C_DOUBLE).

Result type and attributes

REAL(C_DOUBLE)

Result value

The result is an approximation of the bessel function of the first kind and of order 0.

j1(x)
Purpose

Bessel function of the first kind, order 1.

Class

Pure function

Argument type and attributes

x   A **REAL**(C_DOUBLE).

Result type and attributes

REAL(C_DOUBLE)

Result value

The result is an approximation of the bessel function of the first kind and of order 1.

jn(n, x)
Purpose

Bessel function of the first kind, order n.

Class

Pure function

Argument type and attributes

n   An **INTEGER**(C_INT).

x   A **REAL**(C_DOUBLE).

Result type and attributes

REAL(C_DOUBLE)

Result value

The result is an approximation of the bessel function of the first kind and of order **n**.

kill(pid, sig)
Purpose

Sends signal **sig** to the process with ID **pid**.

Class

Function

Argument type and attributes

pid
   An **INTEGER**(C_PID_T).

sig
   An **INTEGER**(C_INT). The value of the corresponding actual argument must equal that of one of the named constants from Table 107 on page 994.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

killpg(pgrp, sig)
Purpose

Sends signal **sig** to the process group with ID **pgrp**.

Class

Function

Argument type and attributes

pgrp
    An **INTEGER**(C_PID_T).

sig
    An **INTEGER**(C_INT). The value of the corresponding actual argument must
    equal that of one of the named constants from Table 107 on page 994.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

lchown(path, owner, group)
Purpose

Changes the owner and group of a symbolic link.

Class

Function

Argument type and attributes

path
    An **INTENT(IN) CHARACTER**(*).

    If the corresponding actual argument is not null-terminated, trailing spaces in
    it are ignored. If trailing spaces in the corresponding actual argument are
    significant, explicitly null-terminate it. If the corresponding actual argument is
    a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you
    can concatenate C_NULL_CHAR to it.

owner
    An **INTEGER**(C_UID_T).

group
    An **INTEGER**(C_GID_T).

Result type and attributes

INTEGER(`C_INT`)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (`errno`).

lfind(key, base, nelp, width, compar)
Purpose

Linear search of an array.

Class

Function

Argument type and attributes

`key`
   A scalar of any type.

`base`
   A contiguous array. The actual argument corresponding to **base** must be of the same type and type parameters as the actual argument corresponding to **key**.

`nelp`
   An **INTEGER**(`C_SIZE_T`). The value of the corresponding actual argument must equal the number of elements in **base**.

`width`
   An **INTEGER**(`C_SIZE_T`). The value of the corresponding actual argument must equal the size of each element in **base**.

`compar`
   A procedure with an interface that conforms to `compar_iface`.

Result type and attributes

TYPE(`C_PTR`)

Result value

If the value of **key** is found in array **base**, the result is a `C_PTR` that points to the first instance of the value of **key** in **base** in row-major order. Otherwise, `C_NULL_PTR` is returned.

Related information
- "lsearch(key, base, nelp, width, compar)" on page 1018
- "bsearch(key, base, nel, width, compar)" on page 997

link(path1, path2)
Purpose

Creates a hard link to a file.

Class

Function

Argument type and attributes

path1
An **INTENT(IN) CHARACTER**(*). The value of the corresponding actual argument must be the name of an existing file.

path2
An **INTENT(IN) CHARACTER**(*). The value of corresponding actual argument is the name of the hard link to be created.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

lsearch(key, base, nelp, width, compar)
Purpose

Linear search and update of an array.

Class

Function

Argument type and attributes

key
A scalar of any type.

base
An **INTENT(INOUT)** contiguous array. The corresponding actual argument must be of the same type and type parameters as the actual argument corresponding to **key**.

nelp
An **INTENT(INOUT) INTEGER**(C_SIZE_T). The value of the corresponding actual argument must equal the number of elements in **base**.

width
An **INTEGER**(C_SIZE_T). The value the corresponding actual argument must equal the size of each element in **base**.

compar
A procedure with an interface that conforms to compar_iface.

Result type and attributes

INTEGER(C_INT)

Result value

If the value of **key** is found in array **base**, the result is a C_PTR that points to the first instance of the value of **key** in **base** in row-major order. Otherwise, the result is a C_PTR that points to a new entry added to the end of **base**.

Related information
- "bsearch(key, base, nel, width, compar)" on page 997
- "lfind(key, base, nelp, width, compar)" on page 1017

malloc(size)
Purpose

Allocates a block of memory.

Class

Function

Argument type and attributes

size   An **INTEGER(C_SIZE_T)**. The value of the corresponding actual argument is the size of the block of memory to allocate in bytes.

Result type and attributes

INTEGER(C_INTPTR_T)

Result value

The result is the address of the allocated memory block or 0 if unsuccessful. The use of **malloc** is discouraged because it can lead to memory leaks. It is preferred to use allocatable variables instead.

This function sets the error number (errno).

Examples

```
USE XLF_POSIX_BINDINGS
IMPLICIT NONE
INTEGER(C_SIZE_T) num_elements
INTEGER i
INTEGER a(100)
POINTER(p, a)

num_elements = 10
p = malloc(C_SIZEOF(a(1)) * num_elements)
IF (p /= 0) THEN
  ! malloc() was successful.
  ! You can access only num_elements of array a
  ! despite what the declaration says.
  a = [(i, i=1, num_elements)]
  PRINT *, a(1:num_elements)
  CALL free(p)
ELSE
  ! malloc() failed.
  CALL perror('')
END IF
END
```

The output is as follows:

1 2 3 4 5 6 7 8 9 10

Related information
- "ALLOCATABLE (Fortran 2003)" on page 285
- "free(ptr)" on page 1002
- "calloc(nelem, elsize)" on page 998
- "realloc(ptr, size)" on page 1024

mkdir(path, mode)
Purpose

Creates a directory.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

mode
> An **INTEGER**(C_MODE_T). The value of the corresponding actual argument must be one or more of the named constants from Table 100 on page 990.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

mkfifo(path, mode)
Purpose

Creates a FIFO special file.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are

significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

mode
> An **INTEGER**(C_MODE_T). The value of the corresponding actual argument must be one or more of the named constants from Table 100 on page 990.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

nice(incr)
Purpose

Changes the nice value of a process. The larger the nice value, the lower the scheduling priority of the process.

Class

Function

Argument type and attributes

incr
> An **INTEGER**(C_INT). The corresponding actual argument must have a value in the range 0 ≤ **incr** ≤ 2 * NZERO - 1.

Result type and attributes

INTEGER(C_INT)

Result value

The result is the new nice value if successful and -1 if unsuccessful.

This function sets the error number (errno).

pathconf(path, name)
Purpose

Queries the value of a configurable pathname variable.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).

If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

name
>   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_LONG)

Result value

The result is the current value of the configurable pathname variable if successful and -1 if unsuccessful.

This function sets the error number (errno).

Examples

```
USE XLF_POSIX_BINDINGS
TYPE(C_PTR) res
INTEGER(C_LONG) path_len
CHARACTER(:), ALLOCATABLE :: path

path_len = pathconf('.', _PC_PATH_MAX);
IF (path_len /= -1) THEN
  ALLOCATE(character(path_len) :: path)
  res = getcwd(path)
  IF (c_associated(res)) THEN
    PRINT *, 'current directory is ', trim(path)
  END IF
ELSE
  CALL perror('')
END IF
END
```

The output is as follows:
```
current directory is /home/user
```

perror(s)
Purpose

Writes an error message explaining the meaning of the current value of errno to standard error. If s is not empty, it is prepended to the error message.

Class

Subroutine

Argument type and attributes

s   An **INTENT(IN) CHARACTER**(*).

Examples

```
USE XLF_POSIX_BINDINGS
IMPLICIT NONE
INTEGER(C_INTPTR_T) x
x = malloc(-1_c_size_t)
```

```
IF (x == 0) THEN
  CALL perror('malloc failed')
END IF
END
```

The output is as follows:

```
malloc failed: Cannot allocate memory
```

qsort(base, nel, width, compar)
Purpose

Sorts an array using the quick sort algorithm.

Class

Subroutine

Argument type and attributes

base
> An **INTENT(INOUT)** array of any type. The corresponding actual argument is the array to be sorted.

nelp
> An **INTEGER**(C_SIZE_T). The value of the corresponding actual argument must equal the number of elements in **base**.

width
> An **INTEGER**(C_SIZE_T). The value of the corresponding actual argument must equal the size of each element in **base**.

compar
> A procedure with an interface that conforms to compar_iface.

Examples

```
PURE FUNCTION less_than(a, b)
  USE, INTRINSIC :: ISO_C_BINDING
  IMPLICIT NONE
  INTEGER(C_INT), INTENT(IN) :: a, b
  INTEGER(C_INT) less_than
  less_than = (a - b)
END FUNCTION

USE XLF_POSIX_BINDINGS
IMPLICIT NONE
PROCEDURE(compar_iface) less_than
INTEGER(C_INT) a(5)

a = [INTEGER(C_INT) :: 6, 2, 8, 2, 5]
CALL qsort(a, size(a), C_SIZEOF(a(1)), less_than)
PRINT *, a
END
```

The output is as follows:

```
2 2 5 6 8
```

raise(sig)
Purpose

Sends signal **sig** to the calling thread or process.

Class

Function

Argument type and attributes

sig
    An **INTEGER**(C_INT). The value of the corresponding actual argument must
    equal that of one of the named constants from Table 107 on page 994.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and nonzero if unsuccessful.

This function sets the error number (errno).

realloc(ptr, size)
Purpose

Reallocates a block of memory.

Class

Function

Argument type and attributes

ptr
    An **INTEGER**(C_INTPTR_T). The value of the corresponding actual argument
    must have been obtained from **malloc** or **calloc**.

size
    An **INTEGER**(C_SIZE_T). The value of the corresponding actual argument is
    the new size for the memory block in bytes.

Result value

The result is the address of the reallocated memory block or 0 if unsuccessful. The
use of **realloc** is discouraged because it can lead to memory leaks. It is preferred to
use allocatable variables instead.

This function sets the error number errno.

Related information
- "calloc(nelem, elsize)" on page 998
- "malloc(size)" on page 1019

remove(path)
Purpose

Deletes a file. If **path** is a directory name, **remove** is equivalent to **rmdir**. Otherwise,
it is equivalent to **unlink**.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).

> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

Related information
- "rmdir(path)" on page 1026
- "unlink(path)" on page 1036

rename(old, new)
Purpose

Renames a file.

Class

Function

Argument type and attributes

old
> An **INTENT(IN) CHARACTER**(*).

> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

new
> An **INTENT(IN) CHARACTER**(*).

> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

rmdir(path)
Purpose

Removes a directory.

Class

Function

Argument type and attributes

path
>   An **INTENT(IN) CHARACTER**(*).
>
>   If the corresponding actual argument is not null-terminated, trailing spaces in
>   it are ignored. If trailing spaces in the corresponding actual argument are
>   significant, explicitly null-terminate it. If the corresponding actual argument is
>   a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you
>   can concatenate C_NULL_CHAR to it.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

Related information
- "remove(path)" on page 1024
- "unlink(path)" on page 1036

setegid(gid)
Purpose

Sets the effective group ID of the calling process.

Class

Function

Argument type and attributes

gid
>   An **INTEGER**(C_GID_T).

Result type and attributes

INTEGER(`C_INT`)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (`errno`).

seteuid(uid)
Purpose

Sets the effective user ID of the calling process.

Class

Function

Argument type and attributes

uid
> An **INTEGER**(`C_UID_T`).

Result type and attributes

INTEGER(`C_INT`)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (`errno`).

setgid(gid)
Purpose

Sets the real group ID of the calling process.

Class

Function

Argument type and attributes

gid
> An **INTEGER**(`C_GID_T`).

Result type and attributes

INTEGER(`C_INT`)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (`errno`).

setpgid(pid, pgid)
Purpose

Sets the process group ID of a process.

Class

Function

Argument type and attributes

pid
An **INTEGER**(C_PID_T).

pgid
An **INTEGER**(C_PID_T).

Result type and attributes

INTEGER(C_PID_T).

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

setpgrp()
Purpose

Sets the process group ID of the calling process.

Class

Function

Result type and attributes

INTEGER(C_PID_T)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

setregid(rgid, egid)
Purpose

Sets the real and effective group IDs of the calling process.

Class

Function

Argument type and attributes

rgid
 An **INTEGER**(C_GID_T).

egid
 An **INTEGER**(C_GID_T).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

setreuid(ruid, euid)
Purpose

Sets the real and effective user IDs of the calling process.

Class

Function

Argument type and attributes

ruid
 An **INTEGER**(C_UID_T).

euid
 An **INTEGER**(C_UID_T).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

setsid()
Purpose

Creates a session and sets process group ID.

Class

Function

Result type and attributes

INTEGER(C_PID_T)

Result value

If successful, the result is the new process group ID. If unsuccessful, the result is -1.

This function sets the error number (errno).

setuid(uid)
Purpose

Sets the real user ID of the calling process.

Class

Function

Argument type and attributes

uid
    An **INTEGER**(C_UID_T).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

sleep(seconds)
Purpose

Suspends execution of the calling process for **seconds** seconds.

Class

Function

Argument type and attributes

seconds
    An **INTEGER**(4).

Result type and attributes

INTEGER(4)

Result value

If **sleep** returns because the requested time has elapsed, the result is 0. If **sleep** returns because of the delivery of a signal, the result is **seconds** minus the actual time slept.

symlink(path1, path2)
Purpose

Creates a symbolic link named **path2** to the file named in **path1**.

Class

Function

Argument type and attributes

path1
An **INTENT(IN) CHARACTER**(*).

path2
An **INTENT(IN) CHARACTER**(*).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

sync()
Purpose

Requests a file system update.

Class

Subroutine

sysconf(name)
Purpose

Gets the value of a configurable system limit or variable.

Class

Function

Argument type and attributes

name
An **INTEGER**(C_INT). The corresponding actual argument must be the name of a system constant.

Result type and attributes

INTEGER(C_LONG)

Result value

The result is the current value of the requested configurable system variable. If an error occurred, or if the configurable system variable has no limit, the result is -1.

This function sets the error number (errno).

Examples
```
USE XLF_POSIX_BINDINGS
IMPLICIT NONE

INTEGER(C_LONG) arg_max
arg_max = sysconf(_SC_ARG_MAX)
IF (arg_max /= -1) THEN
  PRINT *, 'maximum number of arguments is', arg_max
END IF
END
```

Related information
• "confstr(name, buf)" on page 1001

time(tloc)
Purpose

Gets the value of time since the Epoch calculated in seconds.

Class

Function

Argument type and attributes

tloc
    A **TYPE**(C_PTR). The corresponding actual argument must be either associated with an **INTEGER**(C_TIME_T) variable, or is unassociated. If the actual argument is associated, it is set to the result value of **time**.

Result type and attributes

INTEGER(C_TIME_T)

Result value

The result is the value of time since the Epoch calculated in seconds. If unsuccessful, the result is -1.

toascii(c)
Purpose

Converts to 7-bit US-ASCII.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is IAND(c, z'7f').

Related information
- "IAND(I, J)" on page 628

tolower(c)
Purpose

Converts to lowercase.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is **IACHAR**(lower_c), where lower_c is the lowercase character that corresponds to **ACHAR**(**c**) in the current locale.

Related information
- "ACHAR(I, KIND)" on page 574
- "IACHAR(C, KIND)" on page 627

toupper(c)
Purpose

Converts to uppercase.

Class

Pure function

Argument type and attributes

c   An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is **IACHAR**(upper_c), where upper_c is the uppercase character that corresponds to **ACHAR**(**c**) in the current locale.

Related information
- "ACHAR(I, KIND)" on page 574
- "IACHAR(C, KIND)" on page 627

truncate(path, length)
Purpose

Truncates the file named in **path** to **length** bytes.

Class

Function

Argument type and attributes

path
    An **INTENT(IN) CHARACTER**(*).

    If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

length
    An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

ualarm(useconds, interval)
Purpose

Requests that the operating system generate an alarm signal (SIGALARM) after **useconds** microseconds and, if **interval** is not zero, repeat the alarm every **interval** microseconds.

Class

Function

Argument type and attributes

useconds
    An **INTEGER**(C_USECONDS_T).

interval
   An **INTEGER**(C_USECONDS_T).

Result type and attributes

INTEGER(C_USECONDS_T)

Result value

If there is a pending ualarm request, the result is the number of microseconds remaining until the pending request. Otherwise, the result is 0.

umask(cmask)
Purpose

Sets and gets the file mode creation mask.

Class

Function

Argument type and attributes

cmask
   An **INTEGER**(C_MODE_T). The value of the corresponding actual argument is the new file mode creation mask. The value of the corresponding actual argument must be one or more of the named constants from Table 100 on page 990.

Result type and attributes

INTEGER(C_MODE_T)

Result value

The result is the value of the previous file mode creation mask.

uname(name)
Purpose

Stores identifying information about the current system into **name**.

Class

Function

Argument type and attributes

name
   An **INTENT(OUT)** of type utsname.

Result type and attributes

INTEGER(C_INT)

Result value

The result is non-negative if successful and -1 if unsuccessful.

This function sets the error number errno.

unlink(path)
Purpose

Removes a link to the file named in **path**. If **path** names a symbolic link, the symbolic link is removed, not its target.

Class

Function

Argument type and attributes

path
> An **INTENT(IN) CHARACTER**(*).
>
> If the corresponding actual argument is not null-terminated, trailing spaces in it are ignored. If trailing spaces in the corresponding actual argument are significant, explicitly null-terminate it. If the corresponding actual argument is a literal, you can null-terminate it using the **-qnullterm** option. Otherwise, you can concatenate C_NULL_CHAR to it.

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

usleep(useconds)
Purpose

Suspends execution of the calling process for **useconds** microseconds.

Class

Function

Argument type and attributes

useconds
> An **INTEGER**(C_USECONDS_T).

Result type and attributes

INTEGER(C_INT)

Result value

The result is 0 if successful and -1 if unsuccessful.

This function sets the error number (errno).

wait(stat_loc)
Purpose

Waits for a child process to stop or terminate.

Class

Function

Argument type and attributes

stat_loc
> An **INTENT(OUT) INTEGER**(C_INT). The wait status is assigned to **stat_loc**. It can be interpreted using **wexitstatus**, **wifcontinued**, **wifexited**, **wifsignaled**, **wifstopped**, **wstopsig**, or **wtermsig**.

Result type and attributes

INTEGER(C_PID_T)

Result value

The result is the same as that of wait(-1, stat_loc, 0).

This function sets errno.

Related information
- "wexitstatus(stat_val)" on page 1038
- "wifcontinued(stat_val)" on page 1039
- "wifexited(stat_val)" on page 1039
- "wifsignaled(stat_val)" on page 1039
- "wifstopped(stat_val)" on page 1040
- "wstopsig(stat_val)" on page 1040
- "wtermsig(stat_val)" on page 1041

waitpid(pid, stat_loc, options)
Purpose

Waits for a child process to stop or terminate.

Class

Function

Argument type and attributes

pid
> An **INTEGER**(C_PID_T).

stat_loc
> An **INTENT(OUT) INTEGER**(C_INT). The wait status is assigned to **stat_loc**. It can be interpreted using **wexitstatus**, **wifcontinued**, **wifexited**, **wifsignaled**, **wifstopped**, **wstopsig**, or **wtermsig**.

options
> An **INTEGER**(C_INT). The value of the corresponding actual argument must be one or more of 0, WCONTINUED, WNOHANG, or WUNTRACED.

Result type and attributes

INTEGER(C_PID_T)

Result value

The result is equal to the process ID of the child process whose status is reported in **stat_loc**. If **options** has a value of WNOHANG, and at least one process that is specified by **pid** does not have status ready, the result is 0. Otherwise, the result is -1.

This function sets the error number (errno).

Related information
- "wexitstatus(stat_val)"
- "wifcontinued(stat_val)" on page 1039
- "wifexited(stat_val)" on page 1039
- "wifsignaled(stat_val)" on page 1039
- "wifstopped(stat_val)" on page 1040
- "wstopsig(stat_val)" on page 1040
- "wtermsig(stat_val)" on page 1041

wexitstatus(stat_val)
Purpose

Decodes the process status value in **stat_val** to get the exit status of a process.

Class

Pure function

Argument type and attributes

stat_val
> An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is the exit status that is encoded in **stat_val**.

wifcontinued(stat_val)
Purpose

Decodes the process status value in **stat_val** to query whether a child process was continued.

Class

Pure function

Argument type and attributes

stat_val
An **INTEGER(C_INT)**.

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if the process whose status is in **stat_val** was continued. Otherwise, the result is 0.

wifexited(stat_val)
Purpose

Decodes the process status value in **stat_val** to query whether a child process exited normally.

Class

Pure function

Argument type and attributes

stat_val
An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if the process whose status is in **stat_val** exited normally. Otherwise, the result is 0.

wifsignaled(stat_val)
Purpose

Decodes the process status value in **stat_val** to query whether a child process was terminated because of an uncaught signal.

Class

Pure function

Argument type and attributes

stat_val
    An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if the process whose status is in **stat_val** was terminated due to an uncaught signal. Otherwise, the result is 0.

wifstopped(stat_val)
Purpose

Decodes the process status value in **stat_val** to query whether a child process is currently stopped.

Class

Pure function

Argument type and attributes

stat_val
    An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is nonzero if the process whose status is in **stat_val** is currently stopped. Otherwise, the result is 0.

wstopsig(stat_val)
Purpose

Decodes the process status value in **stat_val** to get the signal number that caused a process to stop.

Class

Pure function

Argument type and attributes

stat_val
    An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is the signal number that caused the process whose status is in **stat_val** to stop.

wtermsig(stat_val)
Purpose

Decodes the process status value in **stat_val** to get the signal number that caused a process to terminate.

Class

Pure function

Argument type and attributes

stat_val
    An **INTEGER**(C_INT).

Result type and attributes

INTEGER(C_INT)

Result value

The result is the signal number that caused the process whose status is in **stat_val** to terminate.

Chapter 22. Extensions for source compatibility (IBM extension)

Record structures

The syntax used for record structures parallels that used for Fortran derived types in most cases. Also, in most cases, the semantics of the two features are parallel. For these reasons, record structures are supported in XL Fortran in a way that makes the two features almost completely interchangeable. Hence,

- An entity of a derived type declared using either syntax can be declared using either a **TYPE** statement or a **RECORD** statement.
- A component of an object of derived type can be selected using either the percent sign or period.
- A derived type declared using the **record structure** declaration has a structure constructor.
- A component of any derived type can be initialized using either the standard "equals" form of initialization or the extended "double slashes" form of initialization.

There are differences, however, as outlined here:

- A standard derived type declaration cannot have a **%FILL** component.
- A **record structure** declaration must not have a **SEQUENCE** or **PRIVATE** statement.
- The **-qalign=struct** option applies only to derived types declared using a **record structure** declaration.
- A derived type declared using a **record structure** declaration may have the same name as an intrinsic type.
- There are differences in the rules for determination of derived types declared using a **record structure** declaration and those declared using a standard derived type declaration.
- A component of a **record structure** cannot have the **PUBLIC** or **PRIVATE** attribute.
- A derived type declared using the **record structure** declaration cannot have the **BIND** attribute or procedures.
- A standard derived type declaration can have zero components, a record structure declaration must have at least one component.

The size of a sequence derived type declared using a standard derived type declaration is equal to the sum of the number of bytes required to hold all of its components.

The size of a sequence derived type declared using a **record structure** declaration is equal to the sum of the number of bytes required to hold all of its components and its padding.

Previously, a numeric sequence structure or character sequence structure that appeared in a common block was treated as if its components were enumerated directly in the common block. Now, that only applies to structures of a type declared using a standard derived type declaration.

Declaring record structures

Declaring a record structure declares a user-defined type in the same way that a standard Fortran derived type definition declares a user-defined type. A type declared using a **record structure** declaration is a derived type. For the most part, rules that apply to derived types declared using the standard Fortran syntax apply to derived types declared using the record structure syntax. In those cases where there is a difference, the difference will be called out by referring to the two as derived types declared using a **record structure** declaration and derived types declared using a standard derived type declaration.

Record structure declarations follow this syntax:

record_structure_dcl:

```
►►──structure_stmt──────────────────────────────►◄

      ┌──────────────────────┐
      ▼                      │
►►─────struct_comp_dcl_item──┴──────────────────►◄

►►──end_structure_stmt───────────────────────────►◄
```

struct_comp_dcl_item:

```
►►──┬──component_def_stmt───┬────────────────────►◄
    ├──record_structure_dcl─┤
    └──parameter_stmt───────┘
```
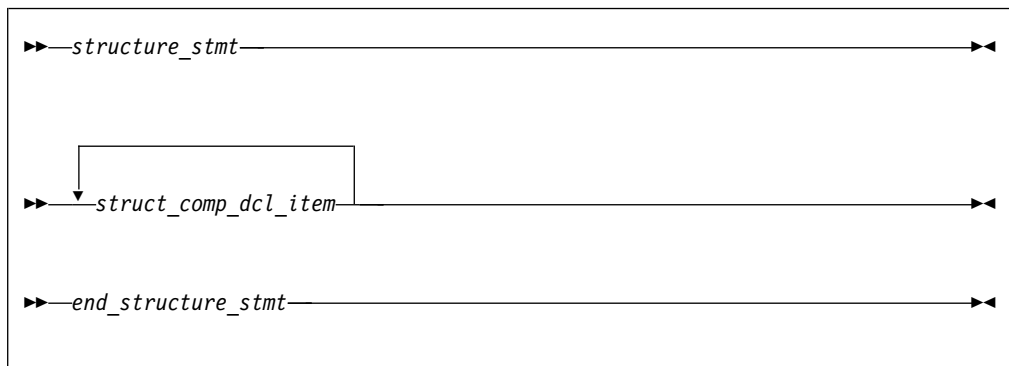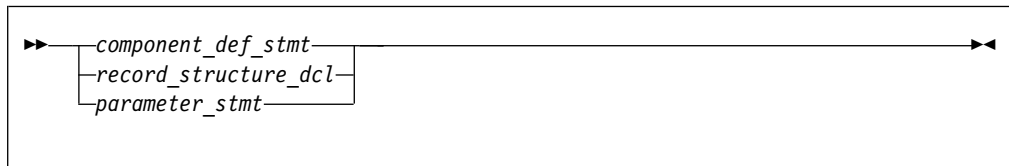
where *component_def_stmt* is a type declaration statement used to define the components of the derived type.

structure_stmt:

```
►►──STRUCTURE──┬─────────────────┬──┬──────────────────┬──►◄
               └─/structure_name/─┘  └─component_dcl_list─┘
```

component_dcl:

```
►►──a──┬──────────────┬──────────────────────────►◄
       └─(-array_spec-)─┘
```
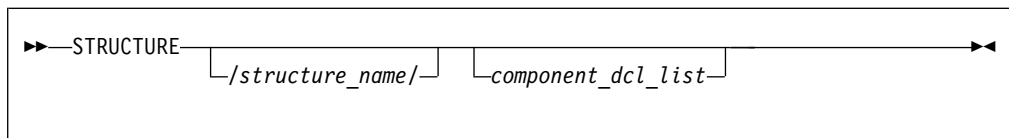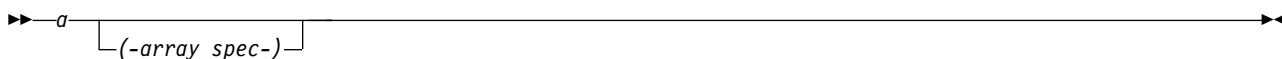
where *a* is an object name.

A structure statement declares the *structure_name* to be a derived type in the scoping unit of the nearest enclosing program unit, interface body or subprogram. The derived type is a local entity of category 1 in that scoping unit.

A structure statement may not specify a *component_dcl_list* unless it is nested in another **record structure** declaration. Likewise, the *structure_name* of a structure statement cannot be omitted unless it is part of a *record_structure_dcl* that is nested in another record structure declaration. A *record_structure_dcl* must have at least one component.

A derived type declared using a **record structure** declaration is a sequence derived type, and is subject to all rules that apply to sequence derived types. A component of a type declared using a **record structure** declaration cannot be of a nonsequence derived type, as is true of sequence derived types declared using standard derived type declarations. A **record structure** declaration cannot contain a **PRIVATE** or **SEQUENCE** statement.

A **record structure** declaration defines a scoping unit. All statements in the *record_structure_dcl* are part of the scoping unit of the record structure declaration, with the exception of any other *record_structure_dcl* contained in the *record_structure_dcl*. These rules are also true of standard derived type declarations, repeated here for clarity.

A *parameter_stmt* in a r*ecord_structure_dcl* declares named constants in the scoping unit of the nearest enclosing program unit, interface body or subprogram. A named constant declared in such a *parameter_stmt* may have the same name as a component declared in the *record_structure_dcl* in which it is contained.

Any components declared on a *structure_stmt* are components of the enclosing derived type, and are local entities of the enclosing structure's scoping unit. The type of such a component is the derived type on whose *structure_stmt* it is declared.

Unlike derived types declared using a standard derived type declaration, a derived type name declared using a **record structure** declaration may be the same as the name of an intrinsic type.

In place of the name of a component, **%FILL** can be used in a *component_def_stm*t in a **record structure** declaration. A **%FILL** component is used as a place-holder to achieve wanted alignment of data in a **record structure** declaration. Initialization cannot be specified for a **%FILL** component. Each instance of **%FILL** in a **record structure** declaration is treated as a unique component name, different from the names of all other components you specified for the type, and different from all other **%FILL** components. **%FILL** is a keyword and is not affected by the **-qmixed** compiler option.

Each instance of a nested structure that has no name is treated as if it had a unique name, different from the names of all other accessible entities.

As an extension to the rules described on derived types thus far, the direct components of a derived type declared using a **record structure** declaration are:
- the components of that type that are not **%FILL** components; and
- the direct components of a derived type component that does not have the **ALLOCATABLE** or **POINTER** attributes and is not a **%FILL** component.

The non-filler ultimate components of a derived type are the ultimate components of the derived type that are also direct components.

An object of a derived type with default initialization can be a member of a common block. You must ensure that a common block is not initialized in more than one scoping unit.

Examples

Example 1: Nested record structure declarations - named and unnamed

```
STRUCTURE /S1/
  STRUCTURE /S2/ A ! A is a component of S1 of type S2
    INTEGER I
  END STRUCTURE
  STRUCTURE B ! B is a component of S1 of unnamed type
    INTEGER J
  END STRUCTURE
END STRUCTURE
RECORD /S1/ R1
RECORD /S2/ R2 ! Type S2 is accessible here.
R2.I = 17
R1.A = R2
R1.B.J = 13
END
```

Example 2: Parameter statement nested in a structure declaration

```
INTEGER I
STRUCTURE /S/
  INTEGER J
  PARAMETER(I=17, J=13) ! Declares I and J in scope of program unit to
                        ! be named constants
END STRUCTURE
INTEGER J ! Confirms implicit typing of named constant J
RECORD /S/ R
R.J = I + J
PRINT *, R.J ! Prints 30
END
```

Example 3: %FILL fields

```
STRUCTURE /S/
  INTEGER I, %FILL, %FILL(2,2), J
  STRUCTURE /S2/ R1, %FILL, R2
    INTEGER I
  END STRUCTURE
END STRUCTURE
RECORD /S/ R
PRINT *, LOC(R%J)-LOC(R%I)  ! Prints 24 with -qintsize=4
PRINT *, LOC(R%R2)-LOC(R%R1) ! Prints 8 with -qintsize=4
END
```

Storage mapping

A derived type declared using a **record structure** declaration is a sequence derived type. In memory, objects of such a type will have the components stored in the order specified. The same is true of objects of a sequence derived type declared using a standard derived type declaration.

The **-qalign** option specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. Both the **[no]4k** and **struct** suboptions

can be specified and are not mutually exclusive. The default setting is
-qalign=no4k:struct=natural. [no]4K is useful primarily in combination with
logical volume I/O and disk striping.

Union and map (IBM extension)

A union declares a group of fields in the enclosing **record structure** that can share
the data area in a program.

Unions and maps follow this syntax:

union_dcl:

```
            ┌──────────────────┐
            │                  │
►►─UNION────▼─union_dcl_item───┴──────────────────────►◄

►►─END UNION──────────────────────────────────────────►◄
```

union_dcl_item:

```
►►─┬─map_dcl────────┬──────────────────────────────────►◄
   └─parameter_stmt─┘
```

map_dcl:

```
          ┌────────────────┐
          │                │
►►─MAP────▼─map_dcl_item───┴────────────────────────────►◄

►►─END MAP──────────────────────────────────────────────►◄
```
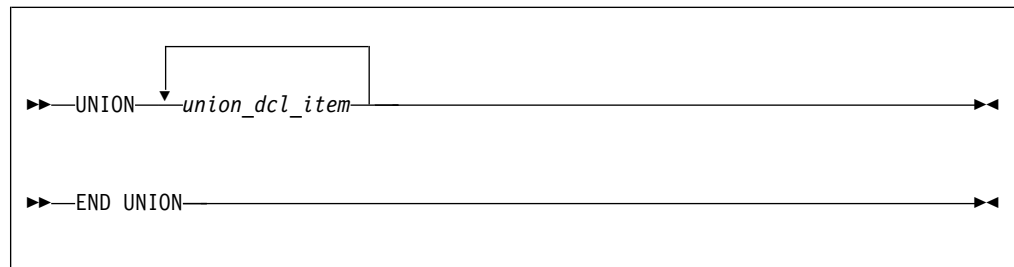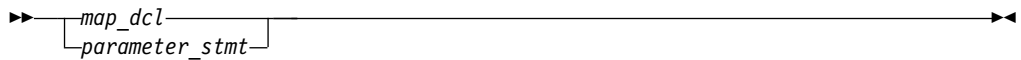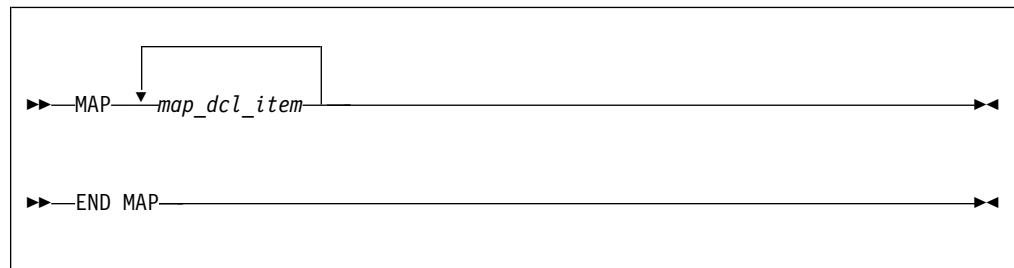
map_dcl_item:

```
►►─┬─struct_comp_dcl_item─┬──────────────────────────────►◄
   └─record_stmt──────────┘
```

struct_comp_dcl_item:

```
         ┌─component_def_stmt──┐
►►───────┼─record_structure_dcl─┼──────────────────────────────►◄
         ├─parameter_stmt──────┤
         └─union_dcl───────────┘
```
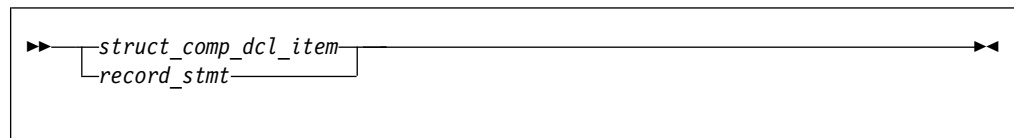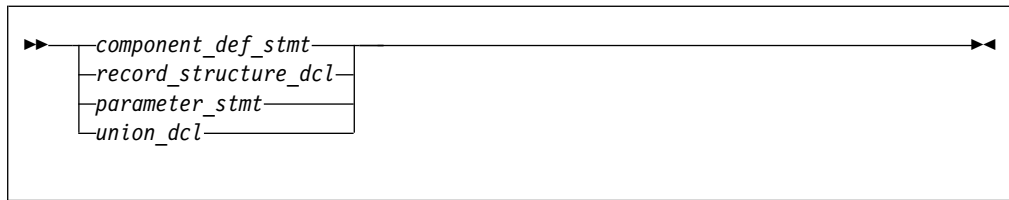
A **union** declaration must be defined in a **record structure**, may be in a **map** declaration, and a **map** declaration must be in a **union** declaration. All declarations in a *map_dcl_item* within a union declaration must be of the same nesting level, regardless of which *map_dcl* they reside in. Therefore, no component name inside a *map_dcl* may appear in any other *map_dcl* on the same level.

A component declared within a **map** declaration must not have a **POINTER** , F2003▶ **PRIVATE**, **PUBLIC**, or **ALLOCATABLE** ◀F2003 attribute.

A record structure with union map must not appear in I/O statements.

The components declared in a **map** declaration share the same storage as the components declared in the other map declarations within a **union** construct. When you assign a value to one component in one **map** declaration, the components in other map declarations that share storage with this component may be affected.

The size of a map is the sum of the sizes of the components declared within it.

The size of the data area established for a **union** declaration is the size of the largest map defined for that union

A *parameter_stmt* in a **map** declaration or **union** construct declares entities in the scoping unit of the nearest enclosing program unit, interface body, or subprogram.

A **%FILL** field in a **map** declaration is used as a place-holder to achieve wanted alignment of data in a record structure. Other non-filler components or part of the components in other map declarations that share the data area with a **%FILL** field are undefined.

If default initialization is specified in *component_def_stmts* in at least one **map** declaration in a **union** declaration, the last occurrence of the initialization becomes the final initialization of the components.

If default initialization is specified in one of the union map declarations in a record structure, a variable of that type that will have its storage class assigned by default will be given
- the static storage class if either the **-qsave=defaultinit** or **-qsave=all** option is specified; or
- the automatic storage class, if the **-qnosave** option is specified.

At any time, only one map is associated with the shared storage. If a component from another map is referenced, the associated map becomes unassociated and its components become undefined. The map referenced will then be associated with the storage.

If a component of *map_dcl* is entirely or partially mapped with the **%FILL** component of the other *map_dcl* in a union, the value of the overlap portion is undefined unless that component is initialized by default initialization or an assignment statement.

Examples

Example 1: The size of the union is equal to the size of the largest map in that union

```
structure /S/
   union
     map
       integer*4  i, j, k
       real*8  r, s, t
     end map
     map
       integer*4  p, q
       real*4  u, v
     end map
   end union         ! Size of the union is 36 bytes.
 end structure
 record /S/ r
```

Example 2: The results of union map are different with different **-qsave** option and suboptions.

```
PROGRAM P
 CALL SUB
 CALL SUB
END PROGRAM P

SUBROUTINE SUB
  LOGICAL, SAVE :: FIRST_TIME = .TRUE.
  STRUCTURE /S/
    UNION
      MAP
        INTEGER I/17/
      END MAP
      MAP
        INTEGER J
      END MAP
    END UNION
  END STRUCTURE
  RECORD /S/ LOCAL_STRUCT
  INTEGER LOCAL_VAR

  IF (FIRST_TIME) THEN
    LOCAL_STRUCT.J = 13
    LOCAL_VAR = 19
    FIRST_TIME = .FALSE.
  ELSE
    ! Prints " 13" if compiled with -qsave or -qsave=all
    ! Prints " 13" if compiled with -qsave=defaultinit
    ! Prints " 17" if compiled with -qnosave
    PRINT *, LOCAL_STRUCT%j
    ! Prints " 19" if compiled with -qsave or -qsave=all
    ! Value of LOCAL_VAR is undefined otherwise
    PRINT *, LOCAL_VAR
  END IF
END SUBROUTINE SUB
```

Example 3: The last occurrence of default initialization in a map declaration within a union structure becomes the final initialization of the component.

```
            structure /st/
              union
                map
                  integer  i /3/, j /4/
                  union
                    map
                      integer  k /8/, l /9/
                    end map
                  end union
                end map
                map
                  integer  a, b
                  union
                    map
                      integer  c /21/
                    end map
                  end union
                end map
              end union
            end structure
            record /st/ R
            print *, R.i, R.j, R.k, R.l      ! Prints "3 4 21 9"
            print *, R.a, R.b, R.c           ! Prints "3 4 21"
            end
```

Example 4: The following program is compiled with **-qintsize=4** and **-qalign=struct=packed**. The components in the union MAP are aligned and packed.

```
            structure /s/
              union
                map
                  integer*2  i /z'1a1a'/, %FILL, j /z'2b2b'/
                end map
                map
                  integer  m, n
                end map
              end union
            end structure
            record /s/ r

            print '(2z6.4)', r.i, r.j      ! Prints "1A1A  2B2B"
            print '(2z10.8)', r.m, r.n     ! Prints "1A1A0000  2B2B0000" however
                                           ! the two bytes in the lower order are
                                           ! not guaranteed.
            r.m = z'abc00cba'              ! Components are initialized by
                                           ! assignment statements.
            r.n = z'02344320'

            print '(2z10.8)', r.m, r.n     ! Prints "ABC00CBA  02344320"
            print '(2z6.4)', r.i, r.j      ! Prints "ABC0  0234"
            end
```

Appendix.

Compatibility across standards

This information is provided for the benefit of users of earlier language standards, such as FORTRAN 77 , who are unfamiliar with more current language standards like Fortran 90, Fortran 95, Fortran 2003, or Fortran 2008, or with XL Fortran.

Except as noted here, the Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 standards are upward-compatible extensions to the preceding Fortran International Standard, ISO 1539-1:1980, informally referred to as FORTRAN 77. Any standard-conforming FORTRAN 77 program remains standard-conforming under the Fortran 90 standard, except as noted under item 4 below regarding intrinsic procedures. Any standard-conforming FORTRAN 77 program remains standard-conforming under the Fortran 95, Fortran 2003, or Fortran 2008 standard, as long as none of the deleted features are used in the program, except as noted under item 4 below regarding intrinsic procedures. The Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 standards restrict the behavior of some features that are processor-dependent in FORTRAN 77. Therefore, a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features may have a different interpretation under the Fortran 90, Fortran 95, Fortran 2003, or Fortran 2008 standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008:

1. FORTRAN 77 permitted a processor to supply more precision derived from a real constant than can be contained in a real datum when the constant is used to initialize a **DOUBLE PRECISION** data object in a **DATA** statement. Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 do not permit this processor-dependent option.

   Previous releases of XL Fortran have been consistent with the Fortran 90 and Fortran 95 behavior.

2. If a named variable that is not in a common block is initialized in a **DATA** statement and does not have the **SAVE** attribute specified, FORTRAN 77 left its **SAVE** attribute processor-dependent. The Fortran 90, Fortan 95, Fortran 2003, and Fortran 2008 standards specify that this named variable has the **SAVE** attribute.

   Previous releases of XL Fortran have been consistent with the Fortran 90 and Fortran 95 behavior.

3. FORTRAN 77 required that the number of characters required by the input list must be less than or equal to the number of characters in the record during formatted input. The Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 standards specify that the input record is logically padded with blanks if there are not enough characters in the record, unless the **PAD='NO'** specifier is indicated in an appropriate **OPEN** statement.

   With XL Fortran, the input record is not padded with blanks if the **noblankpad** suboption of the **-qxlf77** compiler option is specified.

4. The Fortran 90, Fortan 95, Fortran 2003, and Fortran 2008 standards have more intrinsic functions than FORTRAN 77, in addition to a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program may have a different interpretation under Fortran 90, Fortran 95, Fortran 2003, or

Fortran 2008 if it invokes a procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an **EXTERNAL** statement.

With XL Fortran, the **-qextern** compiler option also treats specified names as if they appear in an **EXTERNAL** statement.

5. In Fortran 95, Fortran 2003, and Fortran 2008, for some edit descriptors, a value of 0 for a list item in a formatted output statement will be formatted differently. In addition, the Fortran 95 standard, unlike the FORTRAN 77 standard, specifies how rounding of values will affect the output field form. Therefore, for certain combinations of values and edit descriptors, FORTRAN 77 processors may produce a different output form than Fortran 95 processors.

6. Fortran 95, Fortran 2003, and Fortran 2008 allow a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Fortran 95 changes the behavior of the **SIGN** intrinsic function when the second argument is negative real zero.

7. To distinguish between the Fortran 95 and Fortran 2003 interpretations of signed zeros in the ATAN2(Y,X), LOG(X) and SQRT(X) intrinsics, the **-qxlf2003=signdzerointr** suboption must be used in conjunction with the **-qxlf90=signedzero** option. For the **xlf95** invocation only **-qxlf2003=signdzerointr** needs to be specified since **-qxlf90=signedzero** is on by default. For **xlf2003** none of the options need to be specified since both are on by default. For the **xlf90**, **f77** and **xlf** invocations, both options must be specified in order to have Fortran 2003 standard behaviour.

Fortran 90 compatibility

Except as noted here, the Fortran 95 standard is an upward-compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:1991, informally referred to as Fortran 90. A standard conforming Fortran 90 program that does not use any of the features deleted from the Fortran 95 standard, is a standard conforming Fortran 95 program, as well. The Fortran 90 features that have been deleted from the Fortran 95 standard are the following:

- **ASSIGN** and assigned **GO TO** statements
- **PAUSE** statement
- **DO** control variables and expressions of type real
- **H** edit descriptor
- Branching to an **END IF** statement from outside the **IF** block

Fortran 95 allows a processor to distinguish between a positive and a negative real zero, whereas Fortran 90 did not. Fortran 95 changes the behavior of the **SIGN** intrinsic function when the second argument is negative real zero.

More intrinsic functions appear in the Fortran 95 standard than in the Fortran 90 standard. Therefore, a program that conforms to the Fortran 90 standard may have a different interpretation under the Fortran 95 standard. The different interpretation of the program in Fortran 95 will only occur if the program invokes a procedure that has the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an **EXTERNAL** statement or with an interface body.

Obsolescent features

As the Fortran language evolves, it is only natural that the functionality of some older features are better handled by newer features geared toward today's programming needs. At the same time, the considerable investment in legacy

Fortran code suggests that it would be insensitive to customer needs to decommit any Fortran 90 or FORTRAN 77 features at this time. For this reason, XL Fortran is fully upward compatible with the Fortran 90 and FORTRAN 77 standards. Fortran 95 has removed features that were part of both the Fortran 90 and FORTRAN 77 language standards. However, functionality has not been removed from Fortran 95 as efficient alternatives to the features deleted do exist.

Fortran 95 defines two categories of outmoded features: deleted features and obsolescent features. Deleted features are Fortran 90 or FORTRAN 77 features that are considered to be largely unused and so are not supported in Fortran 95.

Obsolescent features are FORTRAN 77 features that are still frequently used today but whose use can be better delivered by newer features and methods. Although obsolescent features are, by definition, supported in the Fortran 95 standard, some of them may be marked as deleted in the next Fortran standard. Although a processor may still support deleted features as extensions to the language, you may want to take steps now to modify your existing code to use better methods.

Fortran 90 indicates the following FORTRAN 77 features are obsolescent:
- Arithmetic **IF**

  Recommended method: Use the logical **IF** statement, **IF** construct, or **CASE** construct.
- **DO** control variables and expressions of type real

  Recommended method: Use variables and expression of type integer.
- **PAUSE** statement

  Recommended method: Use the **READ** statement.
- Alternate return specifiers

  Recommended method: Evaluate a return code in a **CASE** construct or a computed **GO TO** statement on return from the procedure.

```
! FORTRAN 77

  CALL SUB(A,B,C,*10,*20,*30)

! Fortran 90

  CALL SUB(A,B,C,RET_CODE)
  SELECT CASE (RET_CODE)
    CASE (1)
  .
  .
  .
    CASE (2)
  .
  .
  .
    CASE (3)
  .
  .
  .
  END SELECT
```
- **ASSIGN** and assigned **GO TO** statements

  Recommended method: Use internal procedures.
- Branching to an **END IF** statement from outside the **IF** block

  Recommended method: Branch to the statement that follows the **END IF** statement.
- Shared loop termination and termination on a statement other than **END DO** or **CONTINUE**

  Recommended method: Use an **END DO** or **CONTINUE** statement to terminate each loop.

- **H** edit descriptor

  Recommended method: Use the character constant edit descriptor.

Fortran 95 and Fortran 2003 indicate the following FORTRAN 77 features as obsolescent:
- Arithmetic **IF**

  Recommended method: Use the logical **IF** statement, **IF** construct, or **CASE** construct.
- Alternate return specifiers

  Recommended method: Evaluate a return code in a **CASE** construct or a computed **GO TO** statement on return from the procedure.

  ```
  ! FORTRAN 77

    CALL SUB(A,B,C,*10,*20,*30)

  ! Fortran 90

    CALL SUB(A,B,C,RET_CODE)
    SELECT CASE (RET_CODE)
      CASE (1)
  ⋮
      CASE (2)
  ⋮
      CASE (3)
  ⋮
    END SELECT
  ```
- Shared loop termination and termination on a statement other than **END DO** or **CONTINUE**

  Recommended method: Use an **END DO** or **CONTINUE** statement to terminate each loop.
- Statement functions
- **DATA** statements in executables
- Assumed length character functions
- Fixed source form
- **CHARACTER*** form of declaration

Deleted features

Fortran 2003 and Fortran 95 indicates that the following Fortran 90 and FORTRAN 77 features have been deleted:
- **ASSIGN** and assigned **GO TO** statements
- **PAUSE** statement
- **DO** control variables and expressions of type real
- **H** edit descriptor
- Branching to an **END IF** statement from outside the **IF** block

ASCII and EBCDIC character sets

XL Fortran uses the ASCII character set as its collating sequence.

This table lists the standard ASCII characters in numerical order with the corresponding decimal and hexadecimal values. For convenience in working with programs that use EBCDIC character values, the corresponding information for EBCDIC characters is also included. The table indicates the control characters with "Ctrl-" notation. For example, the horizontal tab (HT) appears as "Ctrl-I", which you enter by simultaneously pressing the Ctrl key and I key.

Table 111. Equivalent characters in the ASCII and EBCDIC character sets

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
0	00	Ctrl-@	NUL	null	NUL	null
1	01	Ctrl-A	SOH	start of heading	SOH	start of heading
2	02	Ctrl-B	STX	start of text	STX	start of text
3	03	Ctrl-C	ETX	end of text	ETX	end of text
4	04	Ctrl-D	EOT	end of transmission	SEL	select
5	05	Ctrl-E	ENQ	enquiry	HT	horizontal tab
6	06	Ctrl-F	ACK	acknowledge	RNL	required new-line
7	07	Ctrl-G	BEL	bell	DEL	delete
8	08	Ctrl-H	BS	backspace	GE	graphic escape
9	09	Ctrl-I	HT	horizontal tab	SPS	superscript
10	0A	Ctrl-J	LF	line feed	RPT	repeat
11	0B	Ctrl-K	VT	vertical tab	VT	vertical tab
12	0C	Ctrl-L	FF	form feed	FF	form feed
13	0D	Ctrl-M	CR	carriage return	CR	carriage return
14	0E	Ctrl-N	SO	shift out	SO	shift out
15	0F	Ctrl-O	SI	shift in	SI	shift in
16	10	Ctrl-P	DLE	data link escape	DLE	data link escape
17	11	Ctrl-Q	DC1	device control 1	DC1	device control 1
18	12	Ctrl-R	DC2	device control 2	DC2	device control 2
19	13	Ctrl-S	DC3	device control 3	DC3	device control 3
20	14	Ctrl-T	DC4	device control 4	RES/ENP	restore/enable presentation
21	15	Ctrl-U	NAK	negative acknowledge	NL	new-line
22	16	Ctrl-V	SYN	synchronous idle	BS	backspace
23	17	Ctrl-W	ETB	end of transmission block	POC	program-operator communications
24	18	Ctrl-X	CAN	cancel	CAN	cancel
25	19	Ctrl-Y	EM	end of medium	EM	end of medium
26	1A	Ctrl-Z	SUB	substitute	UBS	unit backspace
27	1B	Ctrl-[	ESC	escape	CU1	customer use 1
28	1C	Ctrl-\	FS	file separator	IFS	interchange file separator

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
29	1D	Ctrl-]	GS	group separator	IGS	interchange group separator
30	1E	Ctrl-∧	RS	record separator	IRS	interchange record separator
31	1F	Ctrl-_	US	unit separator	IUS/ITB	interchange unit separator / intermediate transmission block
32	20		SP	space	DS	digit select
33	21		!	exclamation mark	SOS	start of significance
34	22		"	straight double quotation mark	FS	field separator
35	23		#	number sign	WUS	word underscore
36	24		$	dollar sign	BYP/INP	bypass/inhibit presentation
37	25		%	percent sign	LF	line feed
38	26		&	ampersand	ETB	end of transmission block
39	27		'	apostrophe	ESC	escape
40	28		(	left parenthesis	SA	set attribute
41	29		)	right parenthesis		
42	2A		*	asterisk	SM/SW	set model switch
43	2B		+	addition sign	CSP	control sequence prefix
44	2C		,	comma	MFA	modify field attribute
45	2D		-	subtraction sign	ENQ	enquiry
46	2E		.	period	ACK	acknowledge
47	2F		/	right slash	BEL	bell
48	30		0			
49	31		1			
50	32		2		SYN	synchronous idle
51	33		3		IR	index return
52	34		4		PP	presentation position
53	35		5		TRN	
54	36		6		NBS	numeric backspace
55	37		7		EOT	end of transmission
56	38		8		SBS	subscript

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
57	39		9		IT	indent tab
58	3A		:	colon	RFF	required form feed
59	3B		;	semicolon	CU3	customer use 3
60	3C		<	less than	DC4	device control 4
61	3D		=	equal	NAK	negative acknowledge
62	3E		>	greater than		
63	3F		?	question mark	SUB	substitute
64	40		@	at symbol	SP	space
65	41		A			
66	42		B			
67	43		C			
68	44		D			
69	45		E			
70	46		F			
71	47		G			
72	48		H			
73	49		I			
74	4A		J		¢	cent
75	4B		K		.	period
76	4C		L		<	less than
77	4D		M		(	left parenthesis
78	4E		N		+	addition sign
79	4F		O		|	logical or
80	50		P		&	ampersand
81	51		Q			
82	52		R			
83	53		S			
84	54		T			
85	55		U			
86	56		V			
87	57		W			
88	58		X			
89	59		Y			
90	5A		Z		!	exclamation mark
91	5B		[	left bracket	$	dollar sign
92	5C		\	left slash	*	asterisk
93	5D		]	right bracket	)	right parenthesis
94	5E		^	hat, circumflex	;	semicolon

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
95	5F		_	underscore	¬	logical not
96	60		`	grave	-	subtraction sign
97	61		a		/	right slash
98	62		b			
99	63		c			
100	64		d			
101	65		e			
102	66		f			
103	67		g			
104	68		h			
105	69		i			
106	6A		j		¦	split vertical bar
107	6B		k		,	comma
108	6C		l		%	percent sign
109	6D		m		_	underscore
110	6E		n		>	greater than
111	6F		o		?	question mark
112	70		p			
113	71		q			
114	72		r			
115	73		s			
116	74		t			
117	75		u			
118	76		v			
119	77		w			
120	78		x			
121	79		y		`	grave
122	7A		z		:	colon
123	7B		{	left brace	#	numbersign
124	7C		|	logical or	@	at symbol
125	7D		}	right brace	'	apostrophe
126	7E		~	similar, tilde	=	equal
127	7F		DEL	delete	"	straight double quotation mark
128	80					
129	81				a	
130	82				b	
131	83				c	
132	84				d	

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
133	85				e	
134	86				f	
135	87				g	
136	88				h	
137	89				i	
138	8A					
139	8B					
140	8C					
141	8D					
142	8E					
143	8F					
144	90					
145	91				j	
146	92				k	
147	93				l	
148	94				m	
149	95				n	
150	96				o	
151	97				p	
152	98				q	
153	99				r	
154	9A					
155	9B					
156	9C					
157	9D					
158	9E					
159	9F					
160	A0					
161	A1				~	similar, tilde
162	A2				s	
163	A3				t	
164	A4				u	
165	A5				v	
166	A6				w	
167	A7				x	
168	A8				y	
169	A9				z	
170	AA					
171	AB					

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
172	AC					
173	AD					
174	AE					
175	AF					
176	B0					
177	B1					
178	B2					
179	B3					
180	B4					
181	B5					
182	B6					
183	B7					
184	B8					
185	B9					
186	BA					
187	BB					
188	BC					
189	BD					
190	BE					
191	BF					
192	C0				{	left brace
193	C1				A	
194	C2				B	
195	C3				C	
196	C4				D	
197	C5				E	
198	C6				F	
199	C7				G	
200	C8				H	
201	C9				I	
202	CA					
203	CB					
204	CC					
205	CD					
206	CE					
207	CF					
208	D0				}	right brace
209	D1				J	
210	D2				K	

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
211	D3				L	
212	D4				M	
213	D5				N	
214	D6				O	
215	D7				P	
216	D8				Q	
217	D9				R	
218	DA					
219	DB					
220	DC					
221	DD					
222	DE					
223	DF					
224	E0				\	left slash
225	E1					
226	E2				S	
227	E3				T	
228	E4				U	
229	E5				V	
230	E6				W	
231	E7				X	
232	E8				Y	
233	E9				Z	
234	EA					
235	EB					
236	EC					
237	ED					
238	EE					
239	EF					
240	F0				0	
241	F1				1	
242	F2				2	
243	F3				3	
244	F4				4	
245	F5				5	
246	F6				6	
247	F7				7	
248	F8				8	
249	F9				9	

Table 111. Equivalent characters in the ASCII and EBCDIC character sets (continued)

Decimal Value	Hex Value	Control Character	ASCII Symbol	Meaning	EBCDIC Symbol	Meaning
250	FA						vertical line
251	FB					
252	FC					
253	FD					
254	FE					
255	FF				EO	eight ones

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL Fortran for Linux.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

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

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA   01886
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2018.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in this product's development: the Electrical Engineering and Computer Sciences Department at the Berkeley campus.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

NVIDIA and CUDA are either registered trademarks or trademarks of NVIDIA Corporation in the United States, other countries, or both.

Glossary

This glossary defines terms that are commonly used in this document. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Terminology* website.

A

abstract interface
> An **ABSTRACT INTERFACE** consists of procedure characteristics and names of dummy arguments. Used to declare the interfaces for procedures and deferred bindings.

abstract type
> A type that has the **ABSTRACT** attribute. A nonpolymorphic object cannot be declared to be of abstract type. A polymorphic object cannot be constructed or allocated to have a dynamic type that is abstract.

active processor
> See *online processor*.

actual argument
> An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

alias
> A single piece of storage that can be accessed through more than a single name. Each name is an alias for that storage.

alphabetic character
> A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as $ and _) allowed by a particular language.

alphanumeric
> Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange
> See *ASCII*.

argument
> An expression that is passed to a function or subroutine. See also *actual argument*, *dummy argument*.

argument association
> The relationship between an actual argument and a dummy argument during the invocation of a procedure.

arithmetic constant
> A constant of type integer, real, or complex.

arithmetic expression
> One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator

A symbol that directs the performance of an arithmetic operation. The intrinsic arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

array An entity that contains an ordered group of scalar data. All objects in an array have the same data type and type parameters.

array declarator

The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension.

array element

A single data item in an array, identified by the array name and one or more subscripts. See also *subscript*.

array name

The name of an ordered set of data items.

array section

A subobject that is an array and is not a structure component.

ASCII The standard code, using a coded character set consisting of 7-bit coded characters (8-bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. See also *Unicode*.

asynchronous

Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals.

assignment statement

An executable statement that defines or redefines a variable based on the result of expression evaluation.

associate name

The name by which a selector of a **SELECT TYPE** or **ASSOCIATE** construct is known within the construct.

assumed-size array

A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

assumed-type object

An entity declared with **TYPE(*)**. An assumed-type object does not have a declared type, and its dynamic type and type parameters are assumed from its corresponding actual argument.

attribute

A property of a data object that may be specified in a type declaration statement, attribute specification statement, or through a default setting.

automatic parallelization

The process by which the compiler attempts to parallelize both explicitly coded **DO** loops and **DO** loops generated by the compiler for array language.

B

base object

An object that is designated by the leftmost *part_name*.

base type

An extensible type that is not an extension of another type.

binary constant

A constant that is made of one or more binary digits (0 and 1).

bind To relate an identifier to another object in a program; for example, to relate an identifier to a value, an address or another identifier, or to associate formal parameters and actual parameters.

binding label

A value of type default character that uniquely identifies how a variable, common block, subroutine, or function is known to a companion processor.

blank common

An unnamed common block.

block data subprogram

A subprogram headed by a **BLOCK DATA** statement and used to initialize variables in named common blocks.

bounds_remapping

Allows a user to view a flat, rank-1 array as a multi-dimensional array.

bss storage

Uninitialized static storage.

busy-wait

The state in which a thread keeps executing in a tight loop looking for more work once it has completed all of its work and there is no new work to do.

byte constant

A named constant that is of type byte.

byte type

A data type representing a one-byte storage area that can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** can be used.

C

C descriptor

A C structure of type `CFI_cdesc_t` that is defined in the ISO_Fortran_binding.h header file.

character constant

A string of one or more alphabetic characters enclosed in apostrophes or double quotation marks.

character expression

A character object, a character-valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

character operator

A symbol that represents an operation, such as concatenation (//), to be performed on character data.

character set

All the valid characters for a programming language or for a computer system.

character string

A sequence of consecutive characters.

character substring

A contiguous portion of a character string.

character type

A data type that consists of alphanumeric characters. See also *data type*.

chunk A subset of consecutive loop iterations.

class A set of types comprised of a base type and all types extended from it.

collating sequence

The sequence in which the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.

comment

A language construct for the inclusion of text in a program that has no effect on the execution of the program.

common block

A storage area that may be referred to by a calling program and one or more subprograms.

compile

To translate a source program into an executable program (an object program).

compiler comment directive

A line in source code that is not a Fortran statement but is recognized and acted on by the compiler.

compiler directive

Source code that controls what XL Fortran does rather than what the user program does.

complex constant

An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

complex number

A number consisting of an ordered pair of real numbers, expressible in the form **a+bi**, where **a** and **b** are real numbers and **i** squared equals -1.

complex type

A data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

component

A constituent of a derived type.

component order

The ordering of the components of a derived type that is used for intrinsic formatted input/output and for structure constructors.

conform

To adhere to a prevailing standard. An executable program conforms to the Fortran 95 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 95 Standard. A program unit conforms to the Fortran 95 Standard if it can be included in an executable program in a manner that allows the executable program to be standard-conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected unit

In XL Fortran, a unit that is connected to a file in one of three ways: explicitly via the **OPEN** statement to a named file, implicitly, or by preconnection.

constant

A data object with a value that does not change. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary). See also *variable*.

construct

A sequence of statements starting with a **SELECT CASE**, **DO**, **IF**, or **WHERE** statement, for example, and ending with the corresponding terminal statement.

contiguous

An array is contiguous if it has array elements in order that are not separated by other data objects. A data object with multiple parts is contiguous if the parts in order are not separated by other data objects.

continuation line

A line that continues a statement beyond its initial line.

control statement

A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as **IF**, or an imperative statement, such as **STOP**.

D

data object

A variable, constant, or subobject of a constant.

data striping

Spreading data across multiple storage devices so that I/O operations can be performed in parallel for better performance. Also known as *disk striping*.

data transfer statement

A **READ**, **WRITE**, or **PRINT** statement.

data type

The properties and internal representation that characterize data and functions. The intrinsic types are integer, real, complex, logical, and character. See also *intrinsic*.

debug line

Allowed only for fixed source form, a line containing source code that is to be used for debugging. Debug lines are defined by a D or X in column 1. The handling of debug lines is controlled by the **-qdlines** and **-qxlines** compiler options.

decimal symbol

The symbol that separates the whole and fractional parts of a real number.

declared type

The type that a data entity is declared to have. May differ from the type during execution (the dynamic type) for polymorphic data entities.

default initialization

The initialization of an object with a value specified as part of a derived type definition.

deferred binding

A binding with the **DEFERRED** attribute. A deferred binding can only appear in an abstract type definition.

definable variable

A variable whose value can be changed by the appearance of its name or designator on the left of an assignment statement.

delimiters

A pair of parentheses or slashes (or both) used to enclose syntactic lists.

denormalized number

An IEEE number with a very small absolute value and lowered precision. A denormalized number is represented by a zero exponent and a non-zero fraction.

derived type

A type whose data have components, each of which is either of intrinsic type or of another derived type.

digit A character that represents a nonnegative integer. For example, any of the numerals from 0 through 9.

directive

A type of comment that provides instructions and information to the compiler.

disk striping

See *data striping*.

DO loop

A range of statements invoked repetitively by a **DO** statement.

DO variable

A variable, specified in a **DO** statement, that is initialized or incremented prior to each occurrence of the statement or statements within a **DO** loop. It is used to control the number of times the statements within the range are executed.

DOUBLE PRECISION constant

A constant of type real with twice the precision of the default real precision.

dummy argument
An entity whose name appears in the parenthesized list following the procedure name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement.

dynamic dimensioning
The process of re-evaluating the bounds of an array each time the array is referenced.

dynamic extent
For a directive, the lexical extent of the directive and all subprograms called from within the lexical extent.

dynamic type
The type of a data entity during execution of a program. The dynamic type of a data entity that is not polymorphic is the same as its declared type.

E

edit descriptor
An abbreviated keyword that controls the formatting of integer, real, or complex data.

effective item
A scalar object resulting from expanding an input/output list.

elemental
Pertaining to an intrinsic operation, procedure or assignment that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

embedded blank
A blank that is surrounded by any other characters.

entity A general term for any of the following: a program unit, procedure, operator, interface block, common block, external unit, statement function, type, named variable, expression, component of a structure, named constant, statement label, construct, or namelist group.

environment variable
A variable that describes the operating environment of the process.

epoch The starting date used for time in POSIX. It is Jan 01 00:00:00 GMT 1970.

executable program
A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, modules, submodules, subprograms and non-Fortran external procedures.

executable statement
A statement that causes an action to be taken by the program; for example, to perform a calculation, test conditions, or alter normal sequential execution.

explicit initialization
The initialization of an object with a value in a data statement initial value list, block data program unit, type declaration statement, or array constructor.

explicit interface
For a procedure referenced in a scoping unit, the property of being an internal procedure, module procedure, intrinsic procedure, external

procedure that has an interface block, recursive procedure reference in its own scoping unit, or dummy procedure that has an interface block.

expression
A sequence of operands, operators, and parentheses. It may be a variable, a constant, or a function reference, or it may represent a computation.

extended-precision constant
A processor approximation to the value of a real number that occupies 16 consecutive bytes of storage.

extended type
An extensible type that is an extension of another type. A type that is declared with the **EXTENDS** attribute.

extensible type
A type from which new types may be derived using the **EXTENDS** attribute. A nonsequence type that does not have the **BIND** attribute.

extension type
A base type is an extension type of itself only. An extended type is an extension type of itself and of all types for which its parent type is an extension.

external file
A sequence of records on an input/output device. See also *internal file*.

external name
The name of a common block, subroutine, or other global procedure, which the linker uses to resolve references from one compilation unit to another.

external procedure
A procedure that is defined by an external subprogram or by a means other than Fortran.

F

field An area in a record used to contain a particular category of data.

file A sequence of records. See also *external file*, *internal file*.

file index
See *i-node*.

final subroutine
A subroutine that is called automatically during finalization.

finalizable
A type that has final subroutines, or that has a finalizable component. An object of finalizable type.

finalization
The process of calling user-defined final subroutines immediately before destroying an object.

floating-point number
A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating-point base to a power indicated by the second numeral.

format

A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files.

To arrange such things as characters, fields, and lines.

formatted data

Data that is transferred between main storage and an input/output device according to a specified format. See also *list-directed* and *unformatted record*.

function

A procedure that returns the value of a single variable or an object and usually has a single exit. See also *intrinsic procedure*, *subprogram*.

G

generic identifier

A lexical token that appears in an **INTERFACE** statement and is associated with all the procedures in an interface block.

H

hard limit

A system resource limit that can only be raised or lowered by using root authority, or cannot be altered because it is inherent in the system or operating environments's implementation. See also *soft limit*.

hexadecimal

Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

hexadecimal constant

A constant, usually starting with special characters, that contains only hexadecimal digits.

high order transformations

A type of optimization that restructures loops and array language.

Hollerith constant

A string of any characters capable of representation by XL Fortran and preceded with *n***H**, where *n* is the number of characters in the string.

host A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module or submodule that contains a module procedure is called the host of the module procedure. A module or submodule is called the host of its descendant submodules.

host association

The process by which an internal subprogram, module subprogram, derived-type definition, or submodule accesses the entities of its host.

host instance

An instance of the host procedure that supplies the host environment of the internal procedure.

I

IPA Interprocedural analysis, a type of optimization that allows optimizations to be performed across procedure boundaries and across calls to procedures in separate source files.

implicit interface

A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not

have an interface block, a dummy procedure that does not have an interface block, or a statement function.

implied DO
An indexing specification (similar to a **DO** statement, but without specifying the word **DO**) with a list of data elements, rather than a set of statements, as its range.

infinity
An IEEE number (positive or negative) created by overflow or division by zero. Infinity is represented by an exponent where all the bits are 1's, and a zero fraction.

inherit
To acquire from a parent. Type parameters, components, or procedure bindings of an extended type that are automatically acquired from its parent type without explicit declaration in the extended type are said to be inherited.

inheritance association
The relationship between the inherited components and the parent component in an extended type.

i-node The internal structure that describes the individual files in the operating system. There is at least one i-node for each file. An i-node contains the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system. Also known as *file index*.

input/output (I/O)
Pertaining to either input or output, or both.

input/output list
A list of variables in an input or output statement specifying the data to be read or written. An output list can also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

integer constant
An optionally signed digit string that contains no decimal point.

interface block
A sequence of statements from an **INTERFACE** statement to its corresponding **END INTERFACE** statement.

interface body
A sequence of statements in an interface block from a **FUNCTION** or **SUBROUTINE** statement to its corresponding **END** statement.

interference
A situation in which two iterations within a **DO** loop have dependencies upon one another.

internal file
A sequence of records in internal storage. See also *external file*.

interprocedural analysis
See *IPA*.

intrinsic
Pertaining to types, operations, assignment statements, and procedures that are defined by Fortran language standards and can be used in any scoping unit without further definition or specification.

intrinsic module

A module that is provided by the compiler and is available to any program.

intrinsic procedure

A procedure that is provided by the compiler and is available to any program.

K

keyword

A statement keyword is a word that is part of the syntax of a statement (or directive) and may be used to identify the statement.

An argument keyword specifies the name of a dummy argument

kind type parameter

A parameter whose values label the available kinds of an intrinsic type or a derived-type parameter that is declared to have the **KIND** attribute.

L

lexical extent

All of the code that appears directly within a directive construct.

lexical token

A sequence of characters with an indivisible interpretation.

link-edit

To create a loadable computer program by means of a linker.

linker A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable.

list-directed

A predefined input/output format that depends on the type, type parameters, and values of the entities in the data list.

literal A symbol or a quantity in a source program that is itself data, rather than a reference to data.

literal constant

A lexical token that directly represents a scalar value of intrinsic type.

load balancing

An optimization strategy that aims at evenly distributing the work load among processors.

logical constant

A constant with a value of either `true` or `false` (or T or F).

logical operator

A symbol that represents an operation on logical expressions:

```
.NOT.     (logical negation)
.AND.     (logical conjunction)
.OR.      (logical union)
.EQV.     (logical equivalence)
.NEQV.    (logical nonequivalence)
.XOR.     (logical exclusive disjunction)
```

loop A statement block that executes repeatedly.

M

_main The default name given to a main program by the compiler if the main program was not named by the programmer.

main program
A program unit to receive control when a program is run. See also *subprogram*.

master thread
The head process of a team of threads.

module
A program unit that contains or accesses definitions to be accessed by other program units.

module procedure interface body
An interface body whose initial statement contains the **MODULE** prefix specifier. A module procedure interface body specifies the interface of a separate module procedure.

module subprogram
A subprogram that is contained in a module or submodule, but is not an internal subprogram. A module subprogram is a function subprogram, a subroutine subprogram, or a separate module subprogram.

mutex A primitive object that provides mutual exclusion between threads. A mutex is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access shared data or run certain application code at a time.

N

NaN (not-a-number)
A symbolic entity encoded in floating-point format that does not correspond to a number. See also *quiet NaN*, *signaling NaN*.

name A lexical token consisting of a letter followed by up to 249 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

named common
A separate, named common block consisting of variables.

namelist group name
The first parameter in the NAMELIST statement that names a list of names to be used in READ, WRITE, and PRINT statements.

negative zero
An IEEE representation where the exponent and fraction are both zero, but the sign bit is 1. Negative zero is treated as equal to positive zero.

nest To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

NEWUNIT value
A negative number that is less than -2 and is unequal to the unit number of any currently connected file. It is a unit value that the runtime library assigns to the variable specified by the **NEWUNIT=** specifier.

nonexecutable statement
A statement that describes the characteristics of a program unit, data, editing information, or statement functions, but does not cause any action to be taken by the program.

nonexisting file
A file that does not physically exist on any accessible storage medium.

normal
A floating-point number that is not denormal, infinity, or NaN.

not-a-number
See *NaN*.

numeric constant
A constant that expresses an integer, real, complex, or byte number.

numeric storage unit
The space occupied by a nonpointer scalar object of type default integer, default real, or default logical.

O

octal Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

octal constant
A constant that is made of octal digits.

one-trip DO-loop
A **DO** loop that is executed at least once, if reached, even if the iteration count is equal to 0. (This type of loop is from FORTRAN 66.)

online processor
In a multiprocessor machine, a processor that has been activated (brought online). The number of online processors is less than or equal to the number of physical processors actually installed in the machine. Also known as *active processor*.

operator
A specification of a particular computation involving one or two operands.

P

pad To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

paging space
Disk storage for information that is resident in virtual memory but is not currently being accessed.

parent component
The component of an entity of extended type that corresponds to its inherited portion.

parent type
The extensible type from which an extended type is derived.

passed-object dummy argument
The dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure was invoked.

PDF See *profile-directed feedback*.

pointee array

An explicit-shape or assumed-size array that is declared in an integer **POINTER** statement or other specification statement.

pointer

A variable that has the **POINTER** attribute. A pointer must not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer-associated.

polymorphic

Able to be of differing types during program execution. An object declared with the **CLASS** keyword is polymorphic.

preconnected file

A file that is connected to a unit at the beginning of execution of the executable program. Standard error, standard input, and standard output are preconnected files (units 0, 5 and 6, respectively).

predefined convention

The implied type and length specification of a data object, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, $, and _ imply type real of length 4.

present

A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure.

primary

The simplest form of an expression: an object, array constructor, structure constructor, function reference, or expression enclosed in parentheses.

procedure

A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains **ENTRY** statements.

procedure binding

See type-bound procedure.

procedure pointer

A procedure entity that has the **EXTERNAL** and **POINTER** attributes. It can be pointer associated with an external procedure, a module procedure, a dummy procedure or another procedure pointer.

profile-directed feedback (PDF)

A type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

program state

The values of user variables at certain points during the execution of a program.

program unit

A main program or subprogram.

pure An attribute of a procedure that indicates there are no side effects.

Q

quiet NaN
> A NaN (not-a-number) value that does not signal an exception. The intent of a quiet NaN is to propagate a NaN result through subsequent computations. See also *NaN*, *signaling NaN*.

R

random access
> An access method in which records can be read from, written to, or removed from a file in any order. See also *sequential access*.

rank The number of dimensions of an array.

real constant
> A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

record A sequence of values that is treated as a whole within a file.

relational expression
> An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression.

relational operator
> The words or symbols used to express a relational condition or a relational expression:

```
.GT.        greater than
.GE.        greater than or equal to
.LT.        less than
.LE.        less than or equal to
.EQ.        equal to
.NE.        not equal to
```

result variable
> The variable that returns the value of a function.

return specifier
> An argument specified for a statement, such as CALL, that indicates to which statement label control should return, depending on the action specified by the subroutine in the RETURN statement.

S

scalar A single datum that is not an array.

> Not having the property of being an array.

scale factor
> A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

scope That part of an executable program within which a lexical token has a single interpretation.

scope attribute
> That part of an executable program within which a lexical token has a single interpretation of a particular named property or entity.

scoping unit
> A derived-type definition.

> A **BLOCK** construct (not including any nested **BLOCK** constructs, derived-type definitions, and interface bodies within it).

An interface body.

A program unit or subprogram, excluding derived-type definitions, **BLOCK** constructs, interface bodies, and subprograms contained within it.

selector
The object that is associated with the associate name in an **ASSOCIATE** construct.

semantics
The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. See also *syntax*.

separate module procedure
A module procedure that is defined by a separate module subprogram or by a function or subroutine subprogram whose initial statement contains the **MODULE** prefix specifier. A separate module procedure defines a module procedure interface body.

separate module subprogram
A module subprogram whose initial statement contains the **MODULE PROCEDURE** statement. A separate module subprogram defines a module procedure interface body.

sequential access
An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file. See also *random access*.

signaling NaN
A NaN (not-a-number) value that signals an invalid operation exception whenever it appears as an operand. The intent of the signaling NaN is to catch program errors, such as using an uninitialized variable. See also *NaN*, *quiet NaN*.

sleep The state in which a thread completely suspends execution until another thread signals it that there is work to do.

SMP See *symmetric multiprocessing*.

soft limit
A system resource limit that is currently in effect for a process. The value of a soft limit can be raised or lowered by a process, without requiring root authority. The soft limit for a resource cannot be raised above the setting of the hard limit. See also *hard limit*.

spill space
The stack space reserved in each subprogram in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

specification statement
A statement that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

stanza A group of lines in a file that together have a common function or define a part of the system. Stanzas are usually separated by blank lines or colons, and each stanza has a name.

statement
A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

statement function
A name, followed by a list of dummy arguments, that is equated with an intrinsic or derived-type expression, and that can be used as a substitute for the expression throughout the program.

statement label
A number made up of one to five digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a **DO**, or to refer to a **FORMAT** statement.

storage association
The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

structure
A scalar data object of derived type.

structure component
The part of a data object of derived-type corresponding to a component of its type.

submodule
A program unit that extends a module or another submodule. A submodule accesses the definitions from its ancestor module or submodules through host association. It might contain definitions to be accessed by its descendant submodules through host association. It might also contain separate module procedures that define module procedure interface bodies declared in its ancestor module or submodules.

subobject
A portion of a named data object that may be referenced or defined independently of other portions. It can be an array element, array section, structure component, or substring.

subprogram
A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram. See also *main program*.

subroutine
A procedure that is invoked by a **CALL** statement or defined assignment statement.

subscript
A subscript quantity or set of subscript quantities enclosed in parentheses and used with an array name to identify a particular array element.

substring
A contiguous portion of a scalar character string. (Although an array section can specify a substring selector, the result is not a substring.)

symmetric multiprocessing (SMP)
A system in which functionally-identical multiple processors are used in parallel, providing simple and efficient load-balancing.

synchronous
> Pertaining to an operation that occurs regularly or predictably with regard to the occurrence of a specified event in another process.

syntax The rules for the construction of a statement. See also *semantics*.

T

target A named data object specified to have the **TARGET** attribute, a data object created by an **ALLOCATE** statement for a pointer, or a subobject of such an object.

thread A stream of computer instructions that is in control of a process. A multithread process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

thread-visible variable
> A variable that can be accessed by more than one thread.

time slice
> An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing unit time is allocated to another task, so a task cannot monopolize processing unit time beyond a fixed limit.

token In a programming language, a character string, in a particular format, that has some defined significance.

trigger constant
> A sequence of characters that identifies comment lines as compiler comment directives.

Type-bound procedure
> A procedure binding in a type definition. The procedure may be referenced by the binding-name via any object of that dynamic type, as a defined operator, by defined assignment, or as part of the finalization process.

type compatible
> All entities are type compatible with other entities of the same type. Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared type.

type declaration statement
> A statement that specifies the type, length, and attributes of an object or function. Objects can be assigned initial values.

type parameter
> A parameter of a data type. **KIND** and **LEN** are the type parameters of intrinsic types. A type parameter of a derived type has either a **KIND** or a **LEN** attribute.
>
> **Note:** The type parameters of a derived type are defined in the derived-type definition.

U

unformatted record
> A record that is transmitted unchanged between internal and external storage.

Unicode
> A universal character encoding standard that supports the interchange,

processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646. See also *ASCII*.

unit   A means of referring to a file to use in input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unsafe option
Any option that could result in undesirable results if used in the incorrect context. Other options may result in very small variations from the default result, which is usually acceptable. Typically, using an unsafe option is an assertion that your code is not subject to the conditions that make the option unsafe.

use association
The association of names in different scoping units specified by a **USE** statement.

V

variable
A data object whose value can be defined and redefined during the execution of an executable program. It may be a named data object, array element, array section, structure component, or substring. Note that in FORTRAN 77, a variable was always scalar and named.

X

XPG4   X/Open Common Applications Environment (CAE) Portability Guide Issue 4; a document which defines the interfaces of the X/Open Common Applications Environment that is a superset of POSIX.1-1990, POSIX.2-1992, and POSIX.2a-1992 containing extensions to POSIX standards from XPG3.

Z

zero-length character
A character object that has a length of 0 and is always defined.

zero-sized array
An array that has a lower bound that is greater than its corresponding upper bound. The array is always defined.

Index

Special characters

IBM ®

Product Number: 5765-J10; 5725-C75

Printed in USA