VMS Software

VSI Fortran Reference Manual

Document Number: DO-DFRTLR-01A

Publication Date: May 2024

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS x86-64 Version 9.2-1 or higher

Software Version: VSI Fortran Version 8.3 for OpenVMS Itanium
VSI Fortran Version 8.3 for OpenVMS Alpha
VSI Fortran Version 8.5 for OpenVMS x86-64

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI Fortran Reference Manual

Legal Notice

Preface

This manual contains the complete description of the VSI Fortran for OpenVMS programming language, which includes Fortran 95 and Fortran 90 features. It contains information about language syntax and semantics, adherence to various Fortran standards, and extensions to those standards.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for experienced applications programmers who have a basic understanding of Fortran concepts and the Fortran 95/90 language, and are using VSI Fortran in either a single-platform or multiplatform environment.

Some familiarity with parallel programming concepts and OpenVMS is helpful. This manual is not a Fortran or programming tutorial.

3. Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 describes language standards, language compatibility, and some features of Fortran 95 and Fortran 90.
- Chapter 2 describes program structure, the Fortran 95/90 character set, and source forms.
- Chapter 3 describes intrinsic and derived data types, constants, variables (scalars and arrays), and substrings.
- Chapter 4 describes expressions and assignment.
- Chapter 5 describes specification statements, which declare the attributes of data objects.
- Chapter 6 describes dynamic allocation.
- Chapter 7 describes constructs and statements that can transfer control within a program.
- Chapter 8 describes program units (including modules), subroutines and functions, and procedure interfaces.
- Chapter 9 summarizes all intrinsic procedures.
- Chapter 10 describes data transfer input/output (I/O) statements.
- Chapter 11 describes the rules for I/O formatting.
- Chapter 12 describes auxiliary I/O statements you can use to perform file operations.
- Chapter 13 describes compilation control statements.
- Chapter 14 describes compiler directives.

- Chapter 15 describes scope and association.
- Appendix A describes obsolescent language features in Fortran 95 and Fortran 90.
- Appendix B describes some statements and language features supported for programs written in older versions of Fortran.
- Appendix C describes the VSI Fortran character sets.
- Appendix D describes data representation models for numeric intrinsic functions.
- Appendix E summarizes VSI Fortran extensions to the Fortran 95 Standard.

4. Related Documents

The following documents are also useful:

- *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/]

  This manual provides information about VSI Fortran program development and the run-time environment. It describes compiling, linking, running, and debugging VSI Fortran programs, run-time error-handling and I/O, performance guidelines, data types, numeric data conversion, calling other procedures and library routines, and compatibility with Fortran 77.

- *VSI Fortran Installation Guide* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-installation-guide/]

  This guide provides information on how to install VSI Fortran.

- OpenVMS documentation set

  This set provides detailed information about components and features of the OpenVMS operating system, such as commands, tools, libraries, and other aspects of the programming environment.

- Standards and Specifications

  The following copyrighted standard and specification documents contain precise descriptions of many of the features found in VSI Fortran:

  - American National Standard Programming Language FORTRAN, ANSI X3.9-1978
  - American National Standard Programming Language Fortran 90, ANSI X3.198-1992

    This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539:1991 (E).

  - American National Standard Programming Language Fortran 95, ANSI X3J3/96-007

    This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539-1:1997 (E).

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. Conventions

The following conventions might be used in this manual:

Ctrl/*x*	A sequence such as **Ctrl/*x*** indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as **PF1** *x* indicates that you must first press and release the key labeled **PF1** and then press and release another key or a pointing device button.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the name of an argument, an attribute, or a reason.
`monospace`	Bold monospace type indicates a command line, command verb, or a qualifier.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.

numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
real	This term refers to all floating-point intrinsic data types as a group.
complex	This term refers to all complex floating-point intrinsic data types as a group.
logical	This term refers to logical intrinsic data types as a group.
integer	This term refers to integer intrinsic data types as a group.
Fortran	This term refers to language information that is common to ANSI FORTRAN-77, ANSI/ISO Fortran 90, ANSI/ISO Fortran 95, and VSI Fortran 90.
Fortran 90	This term refers to language information that is common to ANSI/ISO Fortran 90 and VSI Fortran. For example, a new language feature introduced in the Fortran 90 standard.
Fortran 95	This term refers to language information that is common to ISO Fortran 95 and VSI Fortran. For example, a new language feature introduced in the Fortran 95 standard.
VSI Fortran for OpenVMS	Unless otherwise specified, this term (formerly Compaq Fortran) refers to language information that is common to the Fortran 90 and 95 standards, and any VSI Fortran extensions, running on the OpenVMS operating system. Since the Fortran 90 standard is a superset of the FORTRAN-77 standard, VSI Fortran also supports the FORTRAN-77 standard. VSI Fortran supports all of the deleted features of the Fortran 95 standard.
IA64	This abbreviation refers to the version of the OpenVMS operating system that runs on the Intel ® Itanium ® architecture.

Chapter 1. Overview

1.1. Language Standards Conformance

Fortran 95 includes Fortran 90 and most features of FORTRAN 77. Fortran 90 is a superset that includes FORTRAN 77. VSI Fortran fully supports the Fortran 95, Fortran 90, and FORTRAN 77 Standards.

VSI Fortran conforms to the American National Standard Fortran 95 (ANSI X3J3/96-007) and the American National Standard Fortran 90 (ANSI X3.198-1992).

The ANSI committee X3J3 answers questions of interpretation of Fortran 95 and Fortran 90 language features. Any answers given by the ANSI committee that are related to features implemented in VSI Fortran may result in changes in future releases of the VSI Fortran compiler, even if the changes produce incompatibilities with earlier releases of VSI Fortran.

VSI Fortran also includes support for programs that conform to the previous Fortran standards (ANSI X3.9-1978 and ANSI X3.0-1966), the International Standards Organization standard ISO 1539-1980 (E), the Federal Information Processing Institute standard FIPS 69-1, and the Military Standard 1753 Language Specification.

For More Information:

On VSI Fortran language extensions, see Appendix E.

1.2. Language Compatibility

VSI Fortran is highly compatible with Fortran 77 on supported platforms, and it is substantially compatible with PDP-11.

For More Information:

On language compatibility, compiler options, and program conversion considerations, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

1.3. Fortran 95 Features

Following are some of the Fortran 95 features implemented in VSI Fortran:

- FORALL statement and construct

  In Fortran 90, you could build array values element-by-element by using array constructors and the RESHAPE and SPREAD intrinsics. The Fortran 95 FORALL statement and construct offer an alternative method.

  FORALL allows array elements, array sections, character substrings, or pointer targets to be explicitly specified as a function of the element subscripts. A FORALL construct allows several array assignments to share the same element subscript control.

  FORALL is a generalization of WHERE. They both allow masked array assignment, but FORALL uses element subscripts, while WHERE uses the whole array.

For more information, see Section 4.2.5.

- PURE user-defined procedures

  Pure user-defined procedures do not have side effects, such as changing the value of a variable in a common block. To specify a pure procedure, use the PURE prefix in the function or subroutine statement. Pure functions are allowed in specification statements.

  For more information, see Section 8.5.1.2.

- ELEMENTAL user-defined procedures

  An elemental user-defined procedure is a restricted form of pure procedure. An elemental procedure can be passed an array, which is acted upon one element at a time. To specify an elemental procedure, use the ELEMENTAL prefix in the function or subroutine statement.

  For more information, see Sections Section 8.5.2 and Section 8.5.3.

- CPU_TIME intrinsic subroutine

  This intrinsic subroutine returns a processor-dependent approximation of processor time.

  For more information, see Section 9.4.32.

- NULL intrinsic function

  In Fortran 90, there was no way to assign a null value to the pointer by using a pointer assignment operation. A Fortran 90 pointer had to be explicitly allocated, nullified, or associated with a target during execution before association status could be determined.

  Fortran 95 provides the NULL intrinsic function that can be used to nullify a pointer.

  For more information, see Section 9.4.110.

- Obsolescent features

  Fortran 95 deletes several language features that were obsolescent in Fortran 90, and identifies new obsolescent features.

  VSI Fortran fully supports features deleted in Fortran 95.

  For more information, see Appendix A.

- Derived-type structure default initialization

  In derived-type definitions, you can now specify default initial values for derived-type components.

  For more information, see Section 3.3.2.

- Pointer initialization

  In Fortran 90, there was no way to define the initial value of a pointer. You can now specify default initialization for a pointer.

  For more information, see Sections Section 3.3.1 and Section 3.3.2.

- Automatic deallocation of allocatable arrays

Allocatable arrays whose status is allocated upon routine exit are now automatically deallocated.

For more information, see Section 6.2.1.

- Enhanced CEILING and FLOOR intrinsic functions

  KIND can now be specified for these intrinsic functions.

  For more information, see Sections Section 9.4.22 and Section 9.4.51.

- Enhanced MAXLOC and MINLOC intrinsic functions

  DIM can now be specified for these intrinsic functions.

  For more information, see Sections Section 9.4.95 and Section 9.4.100.

- Enhanced SIGN intrinsic function

  When a specific compiler option is specified, the SIGN function can now distinguish between positive and negative zero if the processor is capable of doing so.

  For more information, see Section 9.4.141.

- Printing of –0.0

  When a specific compiler option is specified, a floating-point value of minus zero (–0.0) can now be printed if the processor can represent it.

- Enhanced WHERE construct

  The WHERE construct has been improved to allow nested WHERE constructs and a masked ELSEWHERE statement. WHERE constructs can now be named.

  For more information, see Section 4.2.4.

- Generic identifier allowed in END INTERFACE statement

  The END INTERFACE statement of an interface block defining a generic routine can now specify a generic identifier.

  For more information, see Section 8.9.2.

- Zero-length formats

  On output, when using I, B, O, Z, and F edit descriptors, the specified value of the field width can be zero. In such cases, the compiler selects the smallest possible positive actual field width that does not result in the field being filled with asterisks (*).

- Comments allowed in namelist input

  Fortran 95 allows comments (beginning with !) in namelist input data.

1.4. Fortran 90 Features

Following are some of the Fortran 90 features implemented in VSI Fortran:

- Free source form

  Fortran 90 provides a free-source form where line positions have no special meaning. There are no reserved columns, trailing comments can appear, and blanks have significance under certain circumstances (for example, `P R O G R A M` is not allowed as an alternative for `PROGRAM`).

  For more information, see Section 2.3.1.

- Modules

  Fortran 90 provides a form of program unit called a module, which is more powerful than (and overcomes limitations of) FORTRAN 77 block data program units.

  A module is a set of declarations that are grouped together under a single, global name. Modules let you encapsulate a set of related items such as data, procedures, and procedure interfaces, and make them available to another program unit.

  Module items can be made private to limit accessibility, provide data abstraction, and to create more secure and portable programs.

  For more information, see Section 8.3.

- User-defined (derived) data types and operators

  Fortran 90 lets you define data types derived from any combination of the intrinsic data types and derived types. The derived-type object can be accessed as a whole, or its individual components can be accessed directly.

  You can extend the intrinsic operators (such as + and *) to user-defined data types, and also define new operators for operands of any type.

  For more information, see Sections Section 3.3 and Section 8.9.4.

- Array operations and features

  In Fortran 90, intrinsic operators and intrinsic functions can operate on array-valued operands (whole arrays or array sections).

  Features for arrays include whole, partial, and masked array assignment (including the WHERE statement for selective assignment), and array-valued constants and expressions. You can create user-defined array-valued functions, use array constructors to specify values of a one-dimensional array, and allocate arrays dynamically (using ALLOCATABLE and POINTER attributes).

  Intrinsic procedures create multidimensional arrays, manipulate arrays, perform operations on arrays, and support computations involving arrays (for example, SUM sums the elements of an array).

  For more information, see Section Section 3.5.2 and Chapter Chapter 9.

- Generic user-defined procedures

  In Fortran 90, user-defined procedures can be placed in generic interface blocks. This allows the procedures to be referenced using the generic name of the block.

  Selection of a specific procedure within the block is based on the properties of the argument, the same way as specific intrinsic functions are selected based on the properties of the argument when generic intrinsic function names are used.

For more information, see Section 8.9.3.

- Pointers

  Fortran 90 pointers are mechanisms that allow dynamic access and processing of data. They allow arrays to be sized dynamically and they allow structures to be linked together.

  A pointer can be of any intrinsic or derived type. When a pointer is associated with a target, it can appear in most expressions and assignments.

  For more information, see Sections Section 5.15 and Section 4.2.3.

- Recursion

  Fortran 90 procedures can be recursive if the keyword RECURSIVE is specified on the FUNCTION or SUBROUTINE statement line.

  For more information, see Chapter 8.

- Interface blocks

  A Fortran 90 procedure can contain an interface block. Interface blocks can be used to do the following:

  - Describe the characteristics of an external or dummy procedure
  - Define a generic name for a procedure
  - Define a new operator (or extend an intrinsic operator)
  - Define a new form of assignment

  For more information, see Section 8.9.

- Extensibility and redundancy

  By using user-defined data types, operators, and meanings, you can extend Fortran to suit your needs. These new data types and their operations can be packaged in modules, which can be used by one or more program units to provide **data abstraction**.

  With the addition of new features and capabilities, some old features become redundant and may eventually be removed from the language. For example, the functionality of the ASSIGN and assigned GO TO statements can be replaced more effectively by internal procedures. The use of certain old features of Fortran can result in less than optimal performance on newer hardware architectures.

  For more information, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/]. For a list of obsolescent features, see Appendix A.

- Additional features for source text

  Lowercase characters are now allowed in source text. A semicolon can be used to separate multiple statements on a single source line. Additional characters have been added to the Fortran character set, and names can have up to 31 characters (including underscores).

  For more information, see Chapter 2.

- Improved facilities for numerical computation

  Intrinsic data types can be specified in a portable way by using a kind type parameter indicating the precision or accuracy required. There are also intrinsic functions that allow you to specify numeric precision and inquire about precision characteristics available on a processor.

  For more information, see Chapters Chapter 3 and Chapter 9.

- Optional procedure arguments

  Procedure arguments can be made optional and keywords can be used when calling procedures, allowing arguments to be listed in any order.

  For more information, see Chapter 8.

- Additional input/output features

  Fortran 90 provides additional keywords for the OPEN and INQUIRE statements. It also permits namelist formatting, and nonadvancing (stream) character-oriented input and output.

  For more information on formatting, see Chapter 10; on OPEN and INQUIRE, see Chapter 12.

- Additional control constructs

  Fortran 90 provides a control construct (CASE) and improves the DO construct. The DO construct can now use CYCLE and EXIT statements, and can have additional (or no) control clauses (for example, WHILE). All control constructs (CASE, DO, and IF) can now be named.

  For more information, see Chapter 7.

- Additional intrinsic procedures

  Fortran 90 provides many more intrinsic procedures than existed in FORTRAN 77. Many of these intrinsics support mathematical operations on arrays, including the construction and transformation of arrays. Bit manipulation and numerical accuracy intrinsics have been added.

  For more information, see Chapter 9.

- Additional specification statements

  The following specification statements are in Fortran 90:

  - INTENT statement (Section 5.10)
  - OPTIONAL statement (Section 5.13)
  - Fortran 90 POINTER statement (Section 5.15)
  - PUBLIC and PRIVATE statements (Section 5.16)
  - TARGET statement (Section 5.18)

- Additional way to specify attributes

  Fortran 90 lets you specify attributes (such as PARAMETER, SAVE, and INTRINSIC) in type declaration statements, as well as in specification statements.

For more information, see Section 5.1.

- Scope and association

  These concepts were implicit in FORTRAN 77, but they are explicitly defined in Fortran 90. In FORTRAN 77, the term scoping unit applies to a program unit, but Fortran 90 expands the term to include internal procedures, interface blocks, and derived-type definitions.

  For more information, see Chapter 15.

Chapter 2. Program Structure, Characters, and Source Forms

2.1. Program Structure

A Fortran program consists of one or more program units. A **program unit** is usually a sequence of statements that define the data environment and the steps necessary to perform calculations; it is terminated by an END statement.

A program unit can be either a main program, an external subprogram, a module, or a block data program unit. An executable program contains one main program, and, optionally, any number of the other kinds of program units. Program units can be separately compiled.

An **external subprogram** is a function or subroutine that is not contained within a main program, a module, or another subprogram. It defines a procedure to be performed and can be invoked from other program units of the Fortran program. Modules and block data program units are not executable, so they are not considered to be procedures. (Modules can contain module procedures, though, which are executable).

Modules contain definitions that can be made accessible to other program units: data and type definitions, definitions of procedures (called **module subprograms**), and **procedure interfaces**. Module subprograms can be either functions or subroutines. They can be invoked by other module subprograms in the module, or by other program units that access the module.

A **block data program unit** specifies initial values for data objects in named common blocks. In Fortran 95/90, this type of program unit can be replaced by a module program unit.

Main programs, external subprograms, and module subprograms can contain **internal subprograms**. The entity that contains the internal subprogram is its **host**. Internal subprograms can be invoked only by their host or by other internal subprograms in the same host. Internal subprograms must not contain internal subprograms.

For More Information:

On program units and procedures, see Chapter 8.

2.1.1. Statements

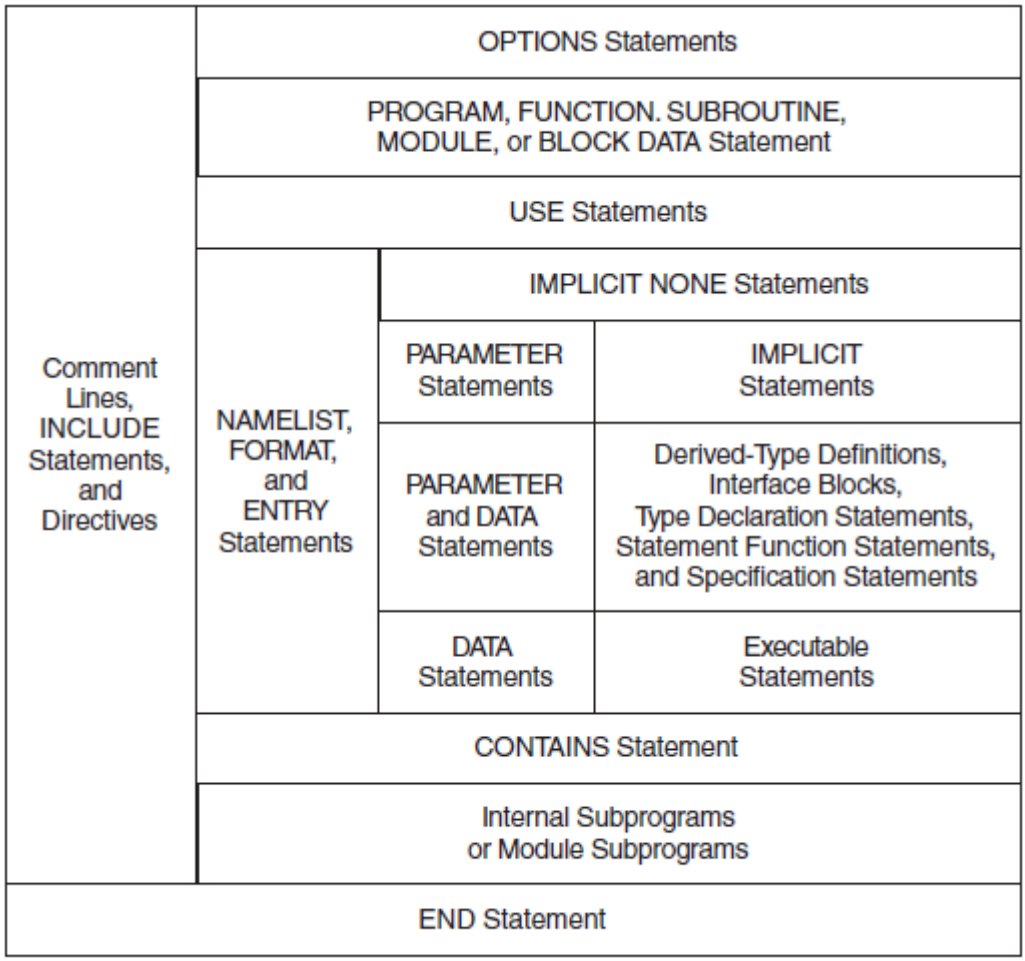
Program statements are grouped into two general classes: executable and nonexecutable. An **executable statement** specifies an action to be performed. A **nonexecutable statement** describes program attributes, such as the arrangement and characteristics of data, as well as editing and data-conversion information.

Order of Statements in a Program Unit

Figure 2.1 shows the required order of statements in a Fortran program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, you can intersperse DATA statements with executable constructs.

Horizontal lines indicate statement types that cannot be interspersed. For example, you cannot intersperse DATA statements with CONTAINS statements.

Figure 2.1. Required Order of Statements



Note that directives and the OPTIONS statement are VSI Fortran language extensions.

PUBLIC and PRIVATE statements are only allowed in the scoping units of modules. In Fortran 95/90, NAMELIST statements can appear only among specification statements. However, VSI Fortran allows them to also appear among executable statements. Table 2.1 shows other statements restricted from different types of scoping units.

Table 2.1. Statements Restricted in Scoping Units

Scoping Unit	Restricted Statements
Main program	ENTRY and RETURN statements
Module[1]	ENTRY, FORMAT, OPTIONAL, and INTENT statements, statement functions, and executable statements
Block data program unit	CONTAINS, ENTRY, and FORMAT statements, interface blocks, statement functions, and executable statements
Internal subprogram	CONTAINS and ENTRY statements
Interface body	CONTAINS, DATA, ENTRY, SAVE, and FORMAT statements, statement functions, and executable statements

[1]The scoping unit of a module does not include any module subprograms that the module contains.

For More Information:

On scoping units, see Section 15.2.

2.1.2. Names

Names identify entities within a Fortran program unit (such as variables, function results, common blocks, named constants, procedures, program units, namelist groups, and dummy arguments). In FORTRAN 77, names were called “symbolic names.”

A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign.

In Fortran 95/90, a name can contain up to 31 characters. VSI Fortran allows names up to 63 characters.

The length of a module name (in MODULE and USE statements) may be restricted by your file system.

In an executable program, the names of the following entities are global and must be unique in the entire program:

- Program units
- External procedures
- Common blocks
- Modules

Examples

The following examples demonstrate valid and invalid names:

Valid	
`NUMBER`	
`FIND_IT`	
`X`	
Invalid	**Explanation**
`5Q`	Begins with a numeral.
`B.4`	Contains a special character other than _ or $.
`_WRONG`	Begins with an underscore.

For More Information:

On the scope of names, see Section 15.2.

2.2. Character Sets

VSI Fortran supports the following characters:

- The Fortran 95/90 character set which consists of the following:

- All uppercase and lowercase letters (A through Z and a through z)
- The numerals 0 through 9
- The underscore (_)
- The following special characters:

Character	Name	Character	Name
Δ or **Tab**	Blank (space) or tab	:	Colon
=	Equal sign	!	Exclamation point
+	Plus sign	"	Quotation mark
–	Minus sign	%	Percent sign
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left parenthesis	<	Less than
)	Right parenthesis	>	Greater than
,	Comma	?	Question mark
.	Period (decimal point)	$	Dollar sign (currency symbol)
'	Apostrophe		

- Other printable characters

  Printable characters include the tab character (09 hex), ASCII characters with codes in the range 20(hex) through 7E(hex), and characters in the DEC Multinational Extension to the ASCII Character Set with codes in the range A1(hex) through FE(hex).

  Printable characters that are not in the Fortran 95/90 character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records.

Uppercase and lowercase letters are treated as equivalent when used to specify program behavior (except in character constants and Hollerith constants).

For More Information:

On the ASCII and DEC Multinational character sets, see Appendix C.

2.3. Source Forms

Within a program, source code can be in free, fixed, or tab form. Fixed or tab forms must not be mixed with free form in the same source program, but different source forms can be used in different source programs.

All source forms allow lowercase characters to be used as an alternative to uppercase characters.

Several characters are indicators in source code (unless they appear within a comment or a Hollerith or character constant). The following are rules for indicators in all source forms:

- Comment indicator

A comment indicator can precede the first statement of a program unit and appear anywhere within a program unit. If the comment indicator appears within a source line, the comment extends to the end of the line.

An all blank line is also a comment line.

Comments have no effect on the interpretation of the program unit.

For more information on comment indicators in free source form, see Section 2.3.1; in fixed and tab source forms, see Section 2.3.2.

- Statement separator

  More than one statement (or partial statement) can appear on a single source line if a statement separator is placed between the statements. The statement separator is a semicolon character (;).

  Consecutive semicolons (with or without intervening blanks) are considered to be one semicolon.

  If a semicolon is the last character on a line, or the last character before a comment, it is ignored.

- Continuation indicator

  A statement can be continued for more than one line by placing a continuation indicator on the line. VSI Fortran allows up to 511 continuation lines in a source program.

  Comments can occur within a continued statement, but comment lines cannot be continued.

  Within a program unit, the END statement cannot be continued, and no other statement in the program unit can have an initial line that appears to be the program unit END statement.

  For more information on continuation indicators in free source form, see Section 2.3.1; in fixed and tab source forms, see Section 2.3.2.

Table 2.2 summarizes characters used as indicators in source forms:

Table 2.2. Indicators in Source Forms

Source Item	Indicator[1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1
Continuation line[2]	&	Free	At the end of the source line
	Any character except zero or blank	Fixed	In column 6
	Any digit except zero	Tab	After the first tab
Statement separator	;	All forms	Between statements on the same line

Source Item	Indicator[1]	Source Form	Position
Statement label	1 to 5 decimal digits	Free	Before a statement
		Fixed	In columns 1 through 5
		Tab	Before the first tab
A debugging statement[3]	D	Fixed	In column 1
		Tab	In column 1

[1]If the character appears in a Hollerith or character constant, it is not an indicator and is ignored.

[2]For all forms, up to 511 continuation lines are allowed.

[3]For fixed and tab forms only.

Source code can be written so that it is useable for all source forms (see Section 2.3.3).

Statement Labels

A **statement label** (or statement number) identifies a statement so that other statements can refer to it, either to get information or to transfer control. A label can precede any statement that is not part of another statement.

A statement label must be one to five decimal digits long; blanks and leading zeros are ignored. An all-zero statement label is invalid, and a blank statement cannot be labeled.

Labeled FORMAT and labeled executable statements are the only statements that can be referred to by other statements. FORMAT statements are referred to only in the format specifier of an I/O statement or in an ASSIGN statement. Two statements within a scoping unit cannot have the same label.

For More Information:

On labels in free source form, see Section 2.3.1; in fixed or tab source form, see Section 2.3.2.

2.3.1. Free Source Form

In free source form, statements are not limited to specific positions on a source line. In Fortran 95/90, a free form source line can contain from 0 to 132 characters. VSI Fortran allows the line to be of any length.

Blank characters are significant in free source form. The following are rules for blank characters:

- Blank characters must not appear in lexical tokens, except within a character context. For example, there can be no blanks between the exponentiation operator **. Blank characters can be used freely between lexical tokens to improve legibility.
- Blank characters must be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels. For example, consider the following statements:

  ```
  INTEGER NUM
  GO TO 40
  20 DO K=1,8
  ```

  The blanks are required after INTEGER, TO, 20, and DO.
- Some adjacent keywords must have one or more blank characters between them. Others do not require any; for example, BLOCK DATA can also be spelled BLOCKDATA. The following list shows which keywords have optional or required blanks:

Optional Blanks	Required Blanks
BLOCK DATA	CASE DEFAULT
DOUBLE COMPLEX	DO WHILE
DOUBLE PRECISION	IMPLICIT *type-specifier*
ELSE IF	IMPLICIT NONE
END BLOCK DATA	INTERFACE ASSIGNMENT
END DO	INTERFACE OPERATOR
END FILE	MODULE PROCEDURE
END FORALL	RECURSIVE FUNCTION
END FUNCTION	RECURSIVE SUBROUTINE
END IF	RECURSIVE *type-specifier* FUNCTION
END INTERFACE	*type-specifier* FUNCTION
END MODULE	*type-specifier* RECURSIVE FUNCTION
END PROGRAM	
END SELECT	
END SUBROUTINE	
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	

For information on statement separators (;) in all forms, see Section 2.3.

Comment Indicator

In free source form, the exclamation point character (!) indicates a comment if it is within a source line, or a comment line if it is the first character in a source line.

Continuation Indicator

In free source form, the ampersand character (&) indicates a continuation line (unless it appears in a Hollerith or character constant, or within a comment). The continuation line is the first noncomment line following the ampersand. Although Fortran 95/90 permits up to 39 continuation lines in free-form programs, VSI Fortran allows up to 511 continuation lines.

The following shows a continued statement:

```
TCOSH(Y) = EXP(Y) + &         ! The initial statement line
           EXP(-Y)            ! A continuation line
```

If the first nonblank character on the next noncomment line is an ampersand, the statement continues at the character following the ampersand. For example, the preceding example can be written as follows:

```
TCOSH(Y) = EXP(Y) + &
```

```
                    & EXP(-Y)
```

If a lexical token must be continued, the first nonblank character on the next noncomment line must be an ampersand followed immediately by the rest of the token. For example:

```
TCOSH(Y) = EXP(Y) + EX&
          &P(-Y)
```

If you continue a character constant, an ampersand must be the first non-blank character of the continued line; the statement continues with the next character following the ampersand. For example:

```
ADVERTISER = "Davis, O'Brien, Chalmers & Peter&
                  &son"
ARCHITECT  = "O'Connor, Emerson, and Davis&
                  & Associates"
```

If the ampersand is omitted on the continued line, the statement continues with the first non-blank character in the continued line. So, in the preceding example, the whitespace before “Associates” would be ignored.

The ampersand cannot be the only nonblank character in a line, or the only nonblank character before a comment; an ampersand in a comment is ignored.

For More Information:

On the general rules for all source forms, see Section 2.3.

2.3.2. Fixed and Tab Source Forms

In Fortran 95, fixed source form is identified as obsolescent.

In fixed and tab source forms, there are restrictions on where a statement can appear within a line.

By default, a statement can extend to character position 72. In this case, any text following position 72 is ignored and no warning message is printed. You can specify a compiler option to extend source lines to character position 132.

Except in a character context, blanks are not significant and can be used freely throughout the program for maximum legibility.

Some Fortran compilers use blanks to pad short source lines out to 72 characters. By default, VSI Fortran does not. If portability is a concern, you can use the concatenation operator to prevent source lines from being padded by other Fortran compilers (see the example in “Continuation Indicator” below) or you can force short source lines to be padded by using a compiler option.

Comment Indicator

In fixed and tab source forms, the exclamation point character (!) indicates a comment if it is within a source line. (It must not appear in column 6 of a fixed form line; that column is reserved for a continuation indicator).

The letter C (or c), an asterisk (*), or an exclamation point (!) indicates a comment line when it appears in column 1 of a source line.

Continuation Indicator

In fixed and tab source forms, a continuation line is indicated by one of the following:

- For fixed form: Any character (except a zero or blank) in column 6 of a source line
- For tab form: Any digit (except zero) after the first tab

The compiler considers the characters following the continuation indicator to be part of the previous line. Although Fortran 95/90 permits up to 19 continuation lines in a fixed-form program, VSI Fortran allows up to 511 continuation lines.

If a zero or blank is used as a continuation indicator, the compiler considers the line to be an initial line of a Fortran statement.

The statement label field of a continuation line must be blank, except in the case of a debugging statement.

When long character or Hollerith constants are continued across lines, portability problems can occur. Use the concatenation operator to avoid such problems. For example:

```
 PRINT *, 'This is a very long character constant '//
+         'which is safely continued across lines'
```

Use this same method when initializing data with long character or Hollerith constants. For example:

```
 CHARACTER*(*) LONG_CONST
 PARAMETER (LONG_CONST = 'This is a very long '//
+ 'character constant which is safely continued '//
+ 'across lines')
 CHARACTER*100 LONG_VAL
 DATA LONG_VAL /LONG_CONST/
```

Hollerith constants must be converted to character constants before using the concatenation method of line continuation.

Debugging Statement Indicator

In fixed and tab source forms, the statement label field can contain a statement label, a comment indicator, or a debugging statement indicator.

The letter D indicates a debugging statement when it appears in column 1 of a source line. The initial line of the debugging statement can contain a statement label in the remaining columns of the statement label field.

If a debugging statement is continued onto more than one line, every continuation line must begin with a D and a continuation indicator.

By default, the compiler treats debugging statements as comments. However, you can specify a compiler option to force the compiler to treat debugging statements as source text to be compiled.

For More Information:

- On the general rules for all source forms, see Section 2.3.

- On statement separators (;) in all forms, see Section 2.3.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the OPTIONS statement, see Section 13.3.
- On statement labels, see Section 2.3.
- On obsolescent features in Fortran 95, see Appendix A.

2.3.2.1. Fixed-Format Lines

In fixed source form, a source line has columns divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character.

The column positions for each field follow:

Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (or 132 with a compiler option)
Sequence number	73 through 80

By default, a sequence number or other identifying information can appear in columns 73 through 80 of any fixed-format line in a VSI Fortran program. The compiler ignores the characters in this field.

If you extend the statement field to position 132, the sequence number field does not exist.

Note

If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

For More Information:

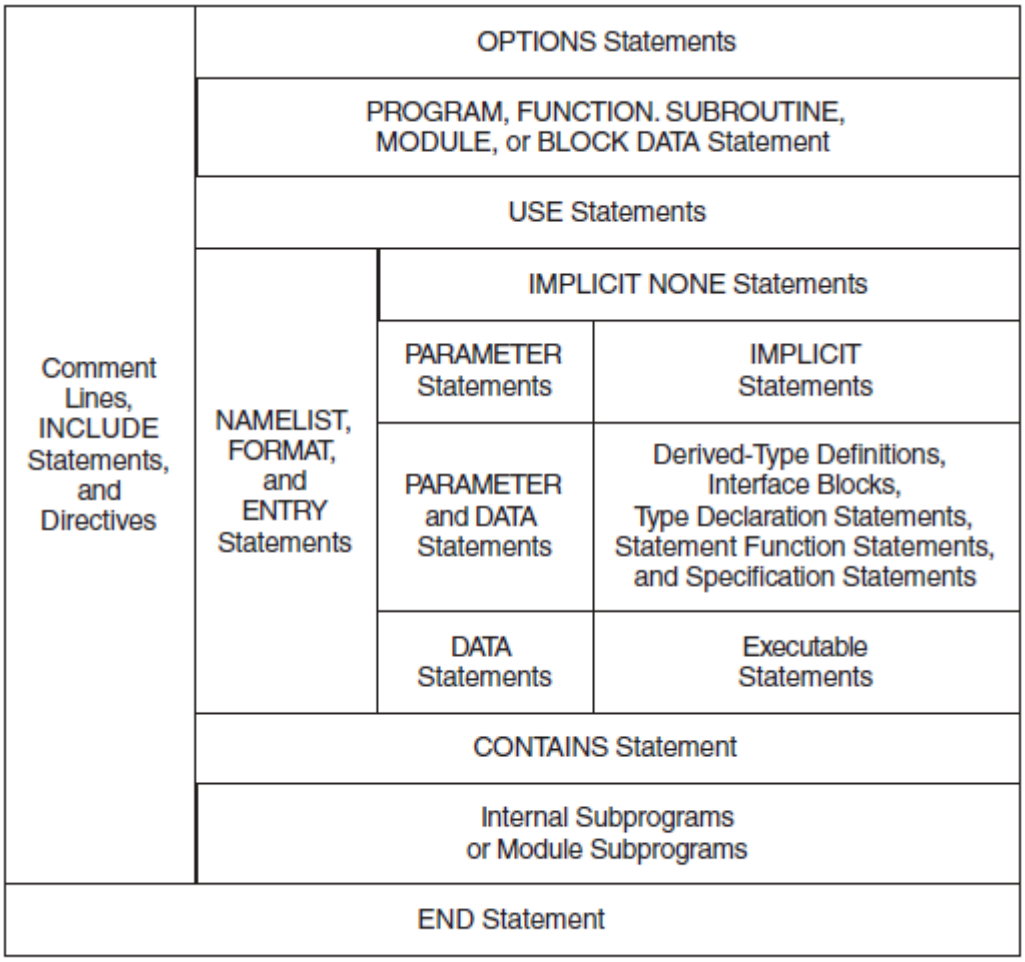
- On the general rules for all source forms, see Section 2.3.
- On the general rules for fixed and tab source forms, see Section 2.3.2.

2.3.2.2. Tab-Format Lines

In tab source form, you can specify a statement label field, a continuation indicator field, and a statement field, but not a sequence number field.

Figure 2.2 shows equivalent source lines coded with tab and fixed source form.

Figure 2.2. Line Formatting Example



The statement label field precedes the first tab character. The continuation indicator field and statement field follow the first tab character.

The continuation indicator is any nonzero digit. The statement field can contain any Fortran statement. A Fortran statement cannot start with a digit.

If a statement is continued, a continuation indicator must be the first character (following the first tab) on the continuation line.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you press the **Tab** key. However, the VSI Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

Note

If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

For More Information:

- On the general rules for all source forms, see Section 2.3.

- On the general rules for fixed and tab source forms, see Section 2.3.2.

2.3.3. Source Code Useable for All Source Forms

To write source code that is useable for all source forms (free, fixed, or tab), follow these rules:

Blanks	Treat as significant (see Section 2.3.1).
Statement labels	Place in column positions 1 through 5 (or before the first tab character).
Statements	Start in column position 7 (or after the first tab character).
Comment indicator	Use only !. Place anywhere *except* in column position 6 (or immediately after the first tab character).
Continuation indicator	Use only &. Place in column position 73 of the initial line and each continuation line, and in column 6 of each continuation line (no tab character can precede the ampersand in column 6).

The following example is valid for all source forms:

```
Column:

12345678...                                                                  73
_______________________________________________________________________________

! Define the user function MY_SIN

      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)                          &
     &           - X**7/FACTOR(7)
      CONTAINS
        INTEGER FUNCTION FACTOR(N)
         FACTOR = 1
         DO 10 I = N, 1, -1
  10     FACTOR = FACTOR * I
        END FUNCTION FACTOR
      END FUNCTION MY_SIN
```

Chapter 3. Data Types, Constants, and Variables

3.1. Overview

Each constant, variable, array, expression, or function reference in a Fortran statement has a data type. The data type of these items can be inherent in their construction, implied by convention, or explicitly declared.

Each **data type** has the following properties:

- A name

  The names of the intrinsic data types are predefined, while the names of derived types are defined in derived-type definitions. Data objects (constants, variables, or parts of constants or variables) are declared using the name of the data type.

- A set of associated values

  Each data type has a set of valid values. Integer and real data types have a range of valid values. Complex and derived types have sets of values that are combinations of the values of their individual components.

- A way to represent constant values for the data type

  A **constant** is a data object with a fixed value that cannot be changed during program execution. The value of a constant can be a numeric value, a logical value, or a character string.

  A constant that does not have a name is a **literal constant**. A literal constant must be of intrinsic type and it cannot be array-valued.

  A constant that has a name is a **named constant**. A named constant can be of any type, including derived type, and it can be array-valued. A named constant has the PARAMETER attribute and is specified in a type declaration statement or PARAMETER statement.

- A set of operations to manipulate and interpret these values

  The data type of a variable determines the operations that can be used to manipulate it. Besides intrinsic operators and operations, you can also define operators and operations.

This chapter contains information on the following topics:

- Intrinsic data types and constants (Section 3.2)
- Derived data types (Section 3.3)
- Binary, octal, hexadecimal, and Hollerith constants (Section 3.4)
- Variables, including arrays (Section 3.5)

For More Information:

- On type declaration statements, see Section 5.1.

- On valid operations for data types, see Section 4.1.
- On defined operations, see Section 4.1.5.
- On ranges for numeric literal constants, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On named constants, see Section 5.14.
- On the PARAMETER attribute and statement, see Section 5.14.

3.2. Intrinsic Data Types

VSI Fortran provides the following intrinsic data types:

- INTEGER (see Section 3.2.1)

  There are four kind parameters for data of type integer:

  - INTEGER([KIND=]1) or INTEGER*1
  - INTEGER([KIND=]2) or INTEGER*2
  - INTEGER([KIND=]4) or INTEGER*4
  - INTEGER([KIND=]8) or INTEGER*8
- REAL (see Section 3.2.2)

  There are three kind parameters for data of type real:

  - REAL([KIND=]4) or REAL*4
  - REAL([KIND=]8) or REAL*8
  - REAL([KIND=]16) or REAL*16
- DOUBLE PRECISION (see Section 3.2.2)

  No kind parameter is permitted for data declared with type DOUBLE PRECISION. This data type is the same as REAL([KIND=]8).
- COMPLEX (see Section 3.2.3)

  There are three kind parameters for data of type complex:

  - COMPLEX([KIND=]4) or COMPLEX*8
  - COMPLEX([KIND=]8) or COMPLEX*16
  - COMPLEX([KIND=]16) or COMPLEX*32
- DOUBLE COMPLEX (see Section 3.2.3)

  No kind parameter is permitted for data declared with type DOUBLE COMPLEX. This data type is the same as COMPLEX([KIND=]8).

- LOGICAL (see Section 3.2.4)

  There are four kind parameters for data of type logical:

  - LOGICAL([KIND=]1) or LOGICAL*1
  - LOGICAL([KIND=]2) or LOGICAL*2
  - LOGICAL([KIND=]4) or LOGICAL*4
  - LOGICAL([KIND=]8) or LOGICAL*8

- CHARACTER (see Section 3.2.5)

  There is one kind parameter for data of type character: CHARACTER([KIND=]1).

- BYTE

  This is a 1-byte value; the data type is equivalent to INTEGER([KIND=]1).

The intrinsic function KIND can be used to determine the kind type parameter of a representation method.

For more portable programs, you should not use the forms INTEGER([KIND=]n) or REAL([KIND=]n). You should instead define a PARAMETER constant using the SELECTED_INT_KIND or SELECTED_REAL_KIND function, whichever is appropriate. For example, the following statements define a PARAMETER constant for an INTEGER kind that has 9 digits:

```
INTEGER, PARAMETER :: MY_INT_KIND = SELECTED_INT_KIND(9)
  ...
INTEGER(MY_INT_KIND) :: J
  ...
```

Note that the syntax separator :: is used in type declaration statements.

The following sections describe the intrinsic data types and forms for literal constants for each type.

For More Information:

- On declaration statements for intrinsic data types, see Section 5.1.1 and Section 5.1.2.
- On operations for intrinsic data types, see Section 4.1.
- On the KIND intrinsic function, see Section 9.4.78.
- On storage requirements for intrinsic data types, see Table 15.2.
- On type declaration statements, see Section 5.1.

3.2.1. Integer Data Types

Integer data types can be specified as follows:

```
INTEGER
INTEGER([KIND=]n)
```

```
INTEGER*n
```

n

Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is **default integer**.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind which holds the constant.

Integer Constants

An **integer constant** is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

Integer constants take the following form:

```
[s]n[n...][ _k]
```

s

Is a sign; required if negative (–), optional if positive (+).

n

Is a decimal digit (0 through 9). Any leading zeros are ignored.

k

Is the optional kind parameter: 1 for INTEGER(1), 2 for INTEGER(2), 4 for INTEGER(4), or 8 for INTEGER(8). It must be preceded by an underscore (_).

An unsigned constant is assumed to be nonnegative.

Integers are expressed in decimal values (base 10) by default. To specify a constant that is not in base 10, use the following syntax:

```
[s][[base] #]nnn...
```

s

Is an optional plus (+) or minus (–) sign.

base

Is any constant from 2 through 36.

If base is omitted but # is specified, the integer is interpreted in base 16. If both base and # are omitted, the integer is interpreted in base 10.

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

Examples

The following examples show valid and invalid integer (base 10) constants:

Valid	
`0`	
`-127`	
`+32123`	
`47_2`	
Invalid	**Explanation**
`9999999999999999999`	Number too large.
`3.14`	Decimal point not allowed; this is a valid REAL constant.
`32,767`	Comma not allowed.
`33_3`	3 is not a valid kind for integers.

The following integers (most of which are not base 10) are all assigned a value equal to 3994575 decimal:

```
I     = 2#11110011111001111001111
m     = 7#45644664
J     = +8#17171717
K     = #3CF3CF
n     = +17#2DE110
L     = 3994575
index = 36#2DM8F
```

You can use integer constants to assign values to data. The following table shows assignments to different data and lists the integer and hexadecimal values in the data:

Fortran Assignment	**Integer Value in the Data**	**Hexadecimal Value in the Data**
LOGICAL(1) X INTEGER(1) X		
X = –128	–128	Z '80'
X = 127	127	Z '7F'
X = 255	–1	Z 'FF'
LOGICAL(2) X INTEGER(2) X		
X = 255	255	Z 'FF'
X = –32768	–32768	Z '8000'
X = 32767	32767	Z '7FFF'
X = 65535	–1	Z 'FFFF'

For More Information:

- On integer constants used in expressions, see Section 4.1.1.
- On the ranges for integer types and kinds, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.2. Real Data Types

Real data types can be specified as follows:

```
REAL
REAL([KIND=]n)
REAL*n
DOUBLE PRECISION
```

n

Is kind 4, 8, or 16.

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is **default real**.

DOUBLE PRECISION is REAL(8). No kind parameter is permitted for data declared with type DOUBLE PRECISION.

3.2.2.1. General Rules for Real Constants

A **real constant** approximates the value of a mathematical real number. The value of the constant can be positive, zero, or negative.

The following is the general form of a real constant with no exponent part:

```
[s]n[n...][ _k]
```

A real constant with an exponent part has one of the following forms:

```
[s]n[n...]E[s]nn...[ _k]
[s]n[n...]D[s]nn...
[s]n[n...]Q[s]nn...
```

s

Is a sign; required if negative (–), optional if positive (+).

n

Is a decimal digit (0 through 9). A decimal point must appear if the real constant has no exponent part.

k

Is the optional kind parameter: 4 for REAL(4), 8 for REAL(8), or 16 for REAL(16). It must be preceded by an underscore (_).

Rules and Behavior

Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting significant digits. For example, in the constant 0.00001234567, all of the nonzero digits, and none of the zeros, are significant. (See the following sections for the number of significant digits each kind type parameter typically has).

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied (for example, 1.0E6 represents the value 1.0 * 10**6).

A real constant with no exponent part and no kind type parameter is (by default) a single-precision (REAL(4)) constant. You can change the default behavior by specifying the compiler option that controls the default real kind.

If the real constant has no exponent part, a decimal point must appear in the string (anywhere before the optional kind parameter). If there is an exponent part, a decimal point is optional in the string preceding the exponent part; the exponent part must not contain a decimal point.

The exponent letter E denotes a single-precision real (REAL(4)) constant, unless the optional kind parameter specifies otherwise. For example, –9.E2_8 is a double-precision constant (which can also be written as –9.D2).

The exponent letter D denotes a double-precision real (REAL(8)) constant.

The exponent letter Q denotes a quad-precision real (REAL(16)) constant.

A minus sign must appear before a negative real constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the exponent letter (E, D, or Q) and a negative exponent, whereas a plus sign is optional between the exponent letter and a positive exponent.

If the real constant includes an exponent letter, the exponent field cannot be omitted, but it can be zero.

To specify a real constant using both an exponent letter and a kind parameter, the exponent letter must be E, and the kind parameter must follow the exponent part.

3.2.2.2. REAL(4) Constants

A single-precision REAL constant occupies four bytes of memory. The number of digits is unlimited, but typically only the leftmost seven digits are significant.

IEEE S_floating format is used.

Examples

The following examples show valid and invalid REAL(4) constants:

Valid	
`3.14159`	
`3.14159_4`	
`621712._4`	
`-.00127`	
`+5.0E3`	
`2E-3_4`	
Invalid	**Explanation**
`1,234,567.`	Commas not allowed.
`325E-47`	Too small for REAL; this is a valid DOUBLE PRECISION constant.
`-47.E47`	Too large for REAL; this is a valid DOUBLE PRECISION constant.
`625._6`	6 is not a valid kind for reals.
`100`	Decimal point missing; this is a valid integer constant.
`$25.00`	Special character not allowed.

For More Information:

- On general rules for real constants, see Section 3.2.2.1.
- On the format and range of REAL(4) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options affecting REAL data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.2.3. REAL(8) or DOUBLE PRECISION Constants

A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.

A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.

IEEE T_floating format is used.

Examples

The following examples show valid and invalid REAL(8) or DOUBLE PRECISION constants:

Valid	
`123456789D+5`	
`123456789E+5_8`	
`+2.7843D00`	
`-.522D-12`	
`2E200_8`	
`2.3_8`	
`3.4E7_8`	
Invalid	**Explanation**
`-.25D0_2`	2 is not a valid kind for reals.
`+2.7182812846182`	No D exponent designator is present; this is a valid single-precision constant.
`1234567890D45`	Too large for D_floating format; valid for G_floating and T_floating format.
`123456789.D400`	Too large for any double-precision format.
`123456789.D-400`	Too small for any double-precision format.

For More Information:

- On general rules for real constants, see Section 3.2.2.1.
- On the format and range of DOUBLE PRECISION (REAL(8)) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options affecting DOUBLE PRECISION data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.2.4. REAL(16) Constants

A REAL(16) constant has more than four times the accuracy of a REAL(4) number, and a greater range.

A REAL(16) constant occupies 16 bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 33 digits are significant.

Examples

The following examples demonstrate valid and invalid REAL(16) constants:

Valid	
`123456789Q4000`	
`-1.23Q-400`	
`+2.72Q0`	
`1.88_16`	
Invalid	**Explanation**
`1.Q5000`	Too large.
`1.Q-5000`	Too small.

For More Information:

- On general rules for real constants, see Section 3.2.2.1.
- On the format and range of REAL(16) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.3. Complex Data Types

Complex data types can be specified as follows:

```
COMPLEX
COMPLEX([KIND=]n)
COMPLEX*s
DOUBLE COMPLEX
```

n

Is kind 4, 8, or 16.

s

Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type **default complex**.

DOUBLE COMPLEX is COMPLEX(8). No kind parameter is permitted for data declared with type DOUBLE COMPLEX.

3.2.3.1. General Rules for Complex Constants

A **complex constant** approximates the value of a mathematical complex number. The constant is a pair of real or integer values, separated by a comma, and enclosed in parentheses. The first constant represents the real part of that number; the second constant represents the imaginary part.

The following is the general form of a complex constant:

```
(c,c)
```

c

Is as follows:

- For COMPLEX(4) constants, *c* is an integer or REAL(4) constant.
- For COMPLEX(8) constants, *c* is an integer, REAL(4) constant, or DOUBLE PRECISION (REAL(8)) constant. At least one of the pair must be DOUBLE PRECISION.
- For COMPLEX(16) constants, *c* is an integer, REAL(4) constant, REAL(8) constant, or REAL(16) constant. At least one of the pair must be REAL(16).

Note that the comma and parentheses are required.

3.2.3.2. COMPLEX(4) Constants

A COMPLEX(4) constant is a pair of integer or single-precision real constants that represent a complex number.

A COMPLEX(4) constant occupies eight bytes of memory and is interpreted as a complex number.

If the real and imaginary part of a complex literal constant are both real, the kind parameter value is that of the part with the greater decimal precision.

The rules for REAL(4) constants apply to REAL(4) constants used in COMPLEX constants. (See Sections Section 3.2.2.1 and Section 3.2.2.2 for the rules on forming REAL(4) constants).

The REAL(4) constants in a COMPLEX constant have IEEE S_floating format.

Examples

The following examples demonstrate valid and invalid COMPLEX(4) constants:

Valid	
`(1.7039,-1.70391)`	
`(44.36_4,-12.2E16_4)`	
`(+12739E3,0.)`	
`(1,2)`	
Invalid	**Explanation**
`(1.23,)`	Missing second integer or single-precision real constant.
`(1.0, 2H12)`	Hollerith constant not allowed.

For More Information:

- On general rules for complex constants, see Section 3.2.3.1.
- On the format and range of COMPLEX (COMPLEX(4)) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options affecting REAL data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.3.3. COMPLEX(8) or DOUBLE COMPLEX Constants

A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(8) or DOUBLE COMPLEX constants. (See Section 3.2.2.1 and Section 3.2.2.3 for the rules on forming DOUBLE PRECISION constants).

The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE T_floating format.

Examples

The following examples demonstrate valid and invalid COMPLEX(8) or DOUBLE COMPLEX constants:

Valid	
`(1.7039,-1.7039D0)`	
`(547.3E0_8,-1.44_8)`	
`(1.7039E0,-1.7039D0)`	
`(+12739D3,0.D0)`	
Invalid	**Explanation**
`(1.23D0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1,1.2)`	Neither constant is DOUBLE PRECISION; this is a valid single-precision constant.

For More Information:

- On general rules for complex constants, see Section 3.2.3.1.
- On the format and range of DOUBLE COMPLEX data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options affecting DOUBLE PRECISION data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.3.4. COMPLEX(16) Constants

A COMPLEX(16) constant is a pair of constants that represents a complex number. One of the pair must be a REAL(16) constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(16) constant occupies 32 bytes of memory and is interpreted as a complex number.

The rules for REAL(16) constants apply to REAL(16) constants used in COMPLEX(16) constants. (See Sections Section 3.2.2.1 and Section 3.2.2.4 for the rules on forming REAL(16) constants.)

The REAL(16) constants in a COMPLEX(16) constant have IEEE X_floating format.

Examples

The following examples demonstrate valid and invalid COMPLEX(16) constants:

Valid	
`(1.7039,-1.7039Q2)`	
`(547.3E0_16,-1.44)`	
`(+12739Q3,0.Q0)`	
Invalid	**Explanation**
`(1.23Q0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1.7039,-1.7039D0)`	Neither constant is REAL(16); this is a valid double-precision constant.

For More Information:

- On general rules for complex constants, see Section 3.2.3.1.
- On the format and range of REAL(16) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options affecting REAL(16) data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.4. Logical Data Types

Logical data types can be specified as follows:

```
LOGICAL
LOGICAL([KIND=]n)
```

```
LOGICAL*n
```

n

Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the logical constant has the kind specified. If no kind parameter is specified, the kind of the constant is **default logical**.

Logical Constants

A **logical constant** represents only the logical values true or false, and takes one of the following forms:

```
.TRUE.[ _k]
.FALSE.[ _k]
```

k

Is the optional kind parameter: 1 for LOGICAL(1), 2 for LOGICAL(2), 4 for LOGICAL(4), or 8 for LOGICAL(8). It must be preceded by an underscore (_).

Logical data type ranges correspond to their comparable integer data type ranges. For example, the LOGICAL(2) range is the same as the INTEGER(2) range.

For More Information:

On integer data type ranges, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.2.5. Character Data Type

The character data type can be specified as follows:

```
CHARACTER
CHARACTER([KIND=]n)
CHARACTER ([LEN=]len)
CHARACTER ([LEN=]len [,[KIND=]n])
CHARACTER (KIND=n [,LEN=len])
CHARACTER*len[,]
```

n

Is kind 1.

len

Is a string length (not a kind). For more information, see Section 5.1.2.

If no kind type parameter is specified, the kind of the constant is **default character**.

Character Constants

A **character constant** is a character string enclosed in delimiters (apostrophes or quotation marks). It takes one of the following forms:

```
[k_ ]’[ch...]’ [C]
```

```
[k_ ]"[ch...]" [C]
```

k

Is the optional kind parameter: 1 (the default). It must be followed by an underscore (_). Note that in character constants, the kind must precede the constant.

ch

Is an ASCII character.

C

Is a C string specifier. C strings can be used to define strings with nonprintable characters. For more information, see Section 3.2.5.1.

Rules and Behavior

The value of a character constant is the string of characters between the delimiters. The value does not include the delimiters, but does include all blanks or tabs within the delimiters.

If a character constant is delimited by apostrophes, use two consecutive apostrophes (`' '`) to place an apostrophe character in the character constant.

Similarly, if a character constant is delimited by quotation marks, use two consecutive quotation marks (`" "`) to place a quotation mark character in the character constant.

The length of the character constant is the number of characters between the delimiters, but two consecutive delimiters are counted as one character.

The length of a character constant must be in the range of 0 to 2000. Each character occupies one byte of memory.

If a character constant appears in a numeric context (such as an expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant.

A zero-length character constant is represented by two consecutive apostrophes or quotation marks.

Examples

The following examples demonstrate valid and invalid character constants:

Valid	
`"WHAT KIND TYPE? "`	
`'TODAY''S DATE IS: '`	
`"The average is: "`	
`''`	
Invalid	**Invalid**
`'HEADINGS`	No trailing apostrophe.
`'Map Number:"`	Beginning delimiter does not match ending delimiter.

For More Information:

On declaring data of type character, see Section 5.1.2.

3.2.5.1. C Strings in Character Constants

String values in the C language are terminated with null characters (CHAR(0)) and can contain nonprintable characters (such as backspace).

Nonprintable characters are specified by escape sequences. An escape sequence is denoted by using the backslash (\) as an escape character, followed by a single character indicating the nonprintable character desired.

This type of string is specified by using a standard string constant followed by the character C. The standard string constant is then interpreted as a C-language constant. Backslashes are treated as escapes, and a null character is automatically appended to the end of the string (even if the string already ends in a null character).

Table 3.1 shows the escape sequences that are allowed in character constants.

Table 3.1. C-Style Escape Sequences

Escape Sequence	Represents
\a or \A	A bell
\b or \B	A backspace
\f or \F	A formfeed
\n or \N	A new line
\r or \R	A carriage return
\t or \T	A horizontal tab
\v or \V	A vertical tab
\x *hh* or \X *hh*	A hexadecimal bit pattern
\ *ooo*	An octal bit pattern
\0	A null character
\ \	A backslash (\)

If a string contains an escape sequence that isn't in this table, the backslash is ignored.

A C string must also be a valid Fortran character constant. If the string is delimited by apostrophes, apostrophes in the string itself must be represented by two consecutive apostrophes (`''`).

For example, the escape sequence `\'string` causes a compiler error because Fortran interprets the apostrophe as the end of the string. The correct form is `\''string`.

If the string is delimited by quotation marks, quotation marks in the string itself must be represented by two consecutive quotation marks (`" "`).

The sequences \ooo and \x *hh* allow any ASCII character to be given as a one- to three-digit octal or a one- to two-digit hexadecimal character code. Each octal digit must be in the range 0 to 7, and each hexadecimal digit must be in the range 0 to F. For example, the C strings `'\010'C` and `'\x08'C` both represent a backspace character followed by a null character.

The C string `'\\abcd'C` is equivalent to the string `'\abcd'` with a null character appended. The string `''C` represents the ASCII null character.

3.2.5.2. Character Substrings

A **character substring** is a contiguous segment of a character string. It takes one of the following forms:

```
v ([e1]:[e2])
a (s [,s] . . . ) ([e1]:[e2])
```

v

Is a character scalar constant, or the name of a character scalar variable or character structure component.

e1

Is a scalar integer (or other numeric) expression specifying the leftmost character position of the substring; the starting point.

e2

Is a scalar integer (or other numeric) expression specifying the rightmost character position of the substring; the ending point.

a

Is the name of a character array.

s

Is a subscript expression.

Both *e1* and *e2* must be within the range 1,2, ..., len, where *len* is the length of the parent character string. If *e1* exceeds *e2*, the substring has length zero.

Rules and Behavior

Character positions within the parent character string are numbered from left to right, beginning at 1.

If the value of the numeric expression e1 or e2 is not of type integer, it is converted to an integer before use (any fractional parts are truncated).

If *e1* is omitted, the default is 1. If *e2* is omitted, the default is *len*. For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

Examples

Consider the following example:

```
CHARACTER*8 C, LABEL
LABEL = 'XVERSUSY'
C = LABEL(2:7)
```

LABEL(2:7) specifies the substring starting with the second character position and ending with the seventh character position of the character variable assigned to LABEL, so C has the value 'VERSUS'.

Consider the following example:

```
TYPE ORGANIZATION
  INTEGER ID
  CHARACTER*35 NAME
END TYPE ORGANIZATION

TYPE(ORGANIZATION) DIRECTOR
CHARACTER*25 BRANCH, STATE(50)
```

The following are valid substrings based on this example:

```
BRANCH(3:15)           ! parent string is a scalar variable
STATE(20) (1:3)        ! parent string is an array element
DIRECTOR%NAME          ! parent string is a structure component
```

Consider the following example:

```
CHARACTER(*), PARAMETER :: MY_BRANCH = "CHAPTER 204"
CHARACTER(3) BRANCH_CHAP
BRANCH_CHAP = MY_BRANCH(9:11) ! parent string is a character constant
```

BRANCH_CHAP is a character string of length 3 that has the value '204'.

For More Information:

- On arrays, see Section 3.5.2.
- On array elements, see Section 3.5.2.2.
- On structure components, see Section 3.3.3.

3.3. Derived Data Types

You can create derived data types from intrinsic data types or previously defined derived types.

A derived type is resolved into “ultimate” components that are either of intrinsic type or are pointers.

The set of values for a specific derived type consists of all possible sequences of component values permitted by the definition of that derived type. Structure constructors are used to specify values of derived types.

Nonintrinsic assignment for derived-type entities must be defined by a subroutine with an ASSIGNMENT interface. Any operation on derived-type entities must be defined by a function with an OPERATOR interface. Arguments and function values can be of any intrinsic or derived type.

For More Information:

- On structure components, see Section 3.3.3.
- On structure constructors, see Section 3.3.4.
- On OPERATOR interfaces, see Section 8.9.4.
- On ASSIGNMENT interfaces, see Section 8.9.5.
- On intrinsic assignment of derived types, see Section 4.2.1.4.

- On record structures, see Section B.12.

3.3.1. Derived-Type Definition

A derived-type definition specifies the name of a user-defined type and the types of its components. It takes the following form:

```
TYPE [ [, access] :: ] name
  component-definition
  [component-definition] . . .
END TYPE [name]
```

access

Is the PRIVATE or PUBLIC keyword. The keyword can only be specified if the derived-type definition is in the specification part of a module.

name

Is the name of the derived type. It must not be the same as the name of any intrinsic type, or the same as the name of a derived type that can be accessed from a module.

component-definition

Is one or more type declaration statements defining the component of derived type.

The first component definition can be preceded by an optional PRIVATE or SEQUENCE statement. (Only one PRIVATE or SEQUENCE statement can appear in a given derived-type definition).

PRIVATE specifies that the components are accessible only within the defining module, even if the derived type itself is public.

SEQUENCE causes the components of the derived type to be stored in the same sequence they are listed in the type definition. If SEQUENCE is specified, all derived types specified in component definitions must be sequence types.

A component definition takes the following form:

```
type [ [, attr] ::] component [(a-spec)] [ *char-len] [init-ex]
```

type

Is a type specifier. It can be an intrinsic type or a previously defined derived type. (If the POINTER attribute follows this specifier, the type can also be any accessible derived type, including the type being defined).

attr

Is an optional POINTER attribute for a pointer component, or an optional DIMENSION attribute for an array component. You can specify one or both attributes. If DIMENSION is specified, it can be followed by an array specification.

The POINTER or DIMENSION attribute can only appear once in a given *component-definition*.

component

Is the name of the component being defined.

a-spec

Is an optional array specification, enclosed in parentheses. If POINTER is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression. For more information on array specifications, see Section 5.1.4.

If the array bounds are not specified here, they must be specified following the DIMENSION attribute.

char-len

Is an optional scalar integer literal constant; it must be preceded by an asterisk (*). This parameter can only be specified if the component is of type CHARACTER.

init-ex

Is an initialization expression or, for pointer components, =>NULL(). This is a Fortran 95 feature.

If *init-ex* is specified, a double colon must appear in the component definition. The equals assignment symbol (=) can only be specified for nonpointer components.

The initialization expression is evaluated in the scoping unit of the type definition.

Rules and Behavior

If a name is specified following the END TYPE statement, it must be the same name that follows TYPE in the derived type statement.

A derived type can be defined only once in a scoping unit. If the same derived-type name appears in a derived-type definition in another scoping unit, it is treated independently.

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.

Two data entities have the same type if they are both declared to be of the same derived type (the derived-type definition can be accessed from a module or a host scoping unit).

If the entities are in different scoping units, they can also have the same derived type if they are declared with reference to different derived-type definitions, and if both derived-type definitions have all of the following:

- The same name
- A SEQUENCE statement (they both have sequence type)
- Components that agree in name, order, and attributes; components cannot be private

For More Information

- On intrinsic data types, see Section 3.2.
- On how to declare variables of derived type, see Section 5.1.3.
- On arrays, see Section 3.5.2.
- On pointers, see Section 5.15.
- On structure components, see Section 3.3.3.

- On default initialization for derived-type components, see Section 3.3.2.
- On alignment of derived-type data components, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.3.2. Default Initialization

Default initialization occurs if initialization appears in a derived-type component definition. (This is a Fortran 95 feature).

The specified initialization of the component will apply even if the definition is PRIVATE.

Default initialization applies to dummy arguments with INTENT(OUT). It does not imply the derived-type component has the SAVE attribute.

Explicit initialization in a type declaration statement overrides default initialization.

To specify default initialization of an array component, use a constant expression that includes one of the following:

- An array constructor
- A single scalar that becomes the value of each array element

Pointers can have an association status of associated, disassociated, or undefined. If no default initialization status is specified, the status of the pointer is undefined. To specify disassociated status for a pointer component, use =>NULL().

Examples

You do not have to specify initialization for each component of a derived type. For example:

```
TYPE REPORT
  CHARACTER (LEN=20) REPORT_NAME
  INTEGER DAY
  CHARACTER (LEN=3) MONTH
  INTEGER :: YEAR = 1995        ! Only component with default
END TYPE REPORT                 !      initialization
```

Consider the following:

```
TYPE (REPORT), PARAMETER :: NOV_REPORT = REPORT ("Sales", 15, "NOV", 1996)
```

In this case, the explicit initialization in the type declaration statement overrides the YEAR component of NOV_REPORT.

The default initial value of a component can also be overridden by default initialization specified in the type definition. For example:

```
TYPE MGR_REPORT
  TYPE (REPORT) :: STATUS = NOV_REPORT
  INTEGER NUM
END TYPE MGR_REPORT

TYPE (MGR_REPORT) STARTUP
```

In this case, the STATUS component of STARTUP gets its initial value from NOV_REPORT, overriding the initialization for the YEAR component.

3.3.3. Structure Components

A reference to a component of a derived-type structure takes the following form:

```
parent [%component [(s-list)]]... %component [(s-list)]
```

`parent`

Is the name of a scalar or array of derived type. The percent sign (%) is called a component selector.

`component`

Is the name of a component of the immediately preceding parent or component.

`s-list`

Is a list of one or more subscripts. If the list contains subscript triplets or vector subscripts, the reference is to an array section.

Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension.

The number of subscripts in any *s-list* must equal the rank of the immediately preceding parent or component.

Rules and Behavior

Each parent or component (except the rightmost) must be of derived type.

The parent or one of the components can have nonzero rank (be an array). Any component to the right of a parent or component of nonzero rank must not have the POINTER attribute.

The rank of the structure component is the rank of the part (parent or component) with nonzero rank (if any); otherwise, the rank is zero. The type and type parameters (if any) of a structure component are those of the rightmost part name.

The structure component must not be referenced or defined before the declaration of the parent object.

If the parent object has the INTENT, TARGET, or PARAMETER attribute, the structure component also has the attribute.

Examples

The following example shows a derived-type definition with two components:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

The following shows how to declare CONTRACT to be of type EMPLOYEE:

```
TYPE(EMPLOYEE) :: CONTRACT
```

Note that both examples started with the keyword TYPE. The first (initial) statement of a derived-type definition is called a derived-type statement, while the statement that declares a derived-type object is called a TYPE statement.

The following example shows how to reference component ID of parent structure CONTRACT:

```
CONTRACT%ID
```

The following example shows a derived type with a component that is a previously defined type:

```
TYPE DOT
  REAL X, Y
END TYPE DOT
....
TYPE SCREEN
  TYPE(DOT) C, D
END TYPE SCREEN
```

The following declares a variable of type SCREEN:

```
TYPE(SCREEN) M
```

Variable M has components M%C and M%D (both of type DOT); M%C has components M%C%X and M%C%Y of type REAL.

The following example shows a derived type with a component that is an array:

```
TYPE CAR_INFO
  INTEGER YEAR
  CHARACTER(LEN=15), DIMENSION(10) :: MAKER
  CHARACTER(LEN=10) MODEL, BODY_TYPE*8
  REAL PRICE
END TYPE
...
TYPE(CAR_INFO) MY_CAR
```

Note that MODEL has a character length of 10, but BODY_TYPE has a character length of 8. You can assign a value to a component of a structure; for example:

```
MY_CAR%YEAR = 1985
```

The following shows an array structure component:

```
MY_CAR%MAKER
```

In the preceding example, if a subscript list (or substring) was appended to MAKER, the reference would not be to an array structure component, but to an array element or section.

Consider the following:

```
MY_CAR%MAKER(2) (4:10)
```

In this case, the component is substring 4 to 10 of the second element of array MAKER.

Consider the following:

```
TYPE CHARGE
  INTEGER PARTS(40)
  REAL LABOR
  REAL MILEAGE
END TYPE CHARGE

TYPE(CHARGE) MONTH
TYPE(CHARGE) YEAR(12)
```

Some valid array references for this type follow:

```
MONTH%PARTS(I)           ! An array element
MONTH%PARTS(I:K)         ! An array section
YEAR(I)%PARTS            ! An array structure component (a whole array)
YEAR(J)%PARTS(I)         ! An array element
YEAR(J)%PARTS(I:K)       ! An array section
YEAR(J:K)%PARTS(I)       ! An array section
YEAR%PARTS(I)            ! An array section
```

The following example shows a derived type with a pointer component that is of the type being defined:

```
TYPE NUMBER
  INTEGER NUM
  TYPE(NUMBER), POINTER :: START_NUM => NULL()
  TYPE(NUMBER), POINTER :: NEXT_NUM  => NULL()
END TYPE
```

A type such as this can be used to construct linked lists of objects of type NUMBER. Note that the pointers are given the default initialization status of disassociated.

The following example shows a private type:

```
TYPE, PRIVATE :: SYMBOL
  LOGICAL TEST
  CHARACTER(LEN=50) EXPLANATION
END TYPE SYMBOL
```

This type is private to the module. The module can be used by another scoping unit, but type SYMBOL is not available.

For More Information

- On references to array elements, see Section 3.5.2.2.
- On references to array sections, see Section 3.5.2.3.
- On examples of derived types in modules, see Section 8.3.

3.3.4. Structure Constructors

A structure constructor lets you specify scalar values of a derived type. It takes the following form:

```
d-name (expr-list)
```

d-name

Is the name of the derived type.

expr-list

Is a list of expressions specifying component values. The values must agree in number and order with the components of the derived type. If necessary, values are converted (according to the rules of assignment), to agree with their corresponding components in type and kind parameters.

Rules and Behavior

A structure constructor must not appear before its derived type is defined.

If a component of the derived type is an array, the shape in the expression list must conform to the shape of the component array.

If a component of the derived type is a pointer, the value in the expression list must evaluate to an object that would be a valid target in a pointer assignment statement. (A constant is not a valid target in a pointer assignment statement).

If all the values in a structure constructor are constant expressions, the constructor is a derived-type constant expression.

Examples

Consider the following derived-type definition:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

This can be used to produce the following structure constructor:

```
EMPLOYEE(3472, "John Doe")
```

The following example shows a type with a component of derived type:

```
TYPE ITEM
  REAL COST
  CHARACTER(LEN=30) SUPPLIER
  CHARACTER(LEN=20) ITEM_NAME
END TYPE ITEM

TYPE PRODUCE
  REAL MARKUP
  TYPE(ITEM) FRUIT
END TYPE PRODUCE
```

In this case, you must use an embedded structure constructor to specify the values of that component; for example:

```
PRODUCE(.70, ITEM (.25, "Daniels", "apple"))
```

For More Information:

On pointer assignment, see Section 4.2.3.

3.4. Binary, Octal, Hexadecimal, and Hollerith Constants

Binary, octal, hexadecimal, and Hollerith constants are nondecimal constants. They have no intrinsic data type, but assume a numeric data type depending on their use.

Fortran 95/90 allows unsigned binary, octal, and hexadecimal constants to be used in DATA statements; the constant must correspond to an integer scalar variable.

In VSI Fortran, binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

3.4.1. Binary Constants

A **binary constant** is an alternative way to represent a numeric constant. A binary constant takes one of the following forms:

```
B’d[d...]’
B"d[d...]"
```

d

Is a binary (base 2) digit (0 or 1).

You can specify up to 256 binary digits in a binary constant. Leading zeros are ignored.

Examples

The following examples demonstrate valid and invalid binary constants:

Valid	
`B'0101110'`	
`B"1"`	
Invalid	**Explanation**
`B'0112'`	The character 2 is invalid.
`B10011'`	No apostrophe after the B.
`"1000001"`	No B before the first quotation mark.

3.4.2. Octal Constants

An **octal constant** is an alternative way to represent numeric constants. An octal constant takes one of the following forms:

```
O’d[d...]’
O"d[d...]"
```

d

Is an octal (base 8) digit (0 through 7).

You can specify up to 256 bits in octal (86 octal digits) constants. Leading zeros are ignored.

Examples

The following examples demonstrate valid and invalid octal constants:

Valid	
`O'07737'`	
`O"1"`	
Invalid	**Explanation**
`O'7782'`	The character 8 is invalid.
`O7772'`	No apostrophe after the O.
`"0737"`	No O before the first quotation mark.

For More Information:

On an alternative form for octal constants, see Section B.7.

3.4.3. Hexadecimal Constants

A **hexadecimal constant** is an alternative way to represent numeric constants. A hexadecimal constant takes one of the following forms:

```
Z'd[d...]'
Z"d[d...]"
```

d

Is a hexadecimal (base 16) digit (0 through 9, or an uppercase or lowercase letter in the range of A to F).

You can specify up to 256 bits in hexadecimal (64 hexadecimal digits) constants. Leading zeros are ignored.

Examples

The following examples demonstrate valid and invalid hexadecimal constants:

Valid	
`Z'AF9730'`	
`Z"FFABC"`	
`Z'84'`	
Invalid	**Explanation**
`Z'999.'`	Decimal not allowed.
`ZF9"`	No quotation mark after the Z.

For More Information:

On an alternative form for hexadecimal constants, see Section B.7.

3.4.4. Hollerith Constants

A **Hollerith constant** is a string of printable ASCII characters preceded by the letter H. Before the H, there must be an unsigned, nonzero default integer constant stating the number of characters in the string (including blanks and tabs).

Hollerith constants are strings of 1 to 2000 characters. They are stored as byte strings, one character per byte.

Examples

The following examples demonstrate valid and invalid Hollerith constants:

Valid	
`16HTODAY'S DATE IS:`	
`1HB`	
`4H ABC`	
Invalid	**Explanation**
`3HABCD`	Wrong number of characters.
`0H`	Hollerith constants must contain at least one character.

3.4.5. Determining the Data Type of Nondecimal Constants

Binary, octal, hexadecimal, and Hollerith constants have no intrinsic data type. These constants assume a numeric data type depending on their use.

When the constant is used with a binary operator (including the assignment operator), the data type of the constant is the data type of the other operand. For example:

Statement	**Data Type of Constant**	**Length of Constant (in bytes)**
`INTEGER(2) ICOUNT`		
`INTEGER(4) JCOUNT`		
`INTEGER(4) N`		
`REAL(8) DOUBLE`		
`REAL(4) RAFFIA, RALPHA`		
`RAFFIA = B'1001100111111010011'`	REAL(4)	4
`RAFFIA = Z'99AF2'`	REAL(4)	4
`RALPHA = 4HABCD`	REAL(4)	4
`DOUBLE = B'11111111111001100110'`	REAL(8)	8
`DOUBLE = Z'FFF99A'`	REAL(8)	8
`DOUBLE = 8HABCDEFGH`	REAL(8)	8
`JCOUNT = ICOUNT + B'011101110111'`	INTEGER(2)	2

Statement	Data Type of Constant	Length of Constant (in bytes)
`JCOUNT = ICOUNT + O'777'`	INTEGER(2)	2
`JCOUNT = ICOUNT + 2HXY`	INTEGER(2)	2
`IF (N .EQ. B'1010100') GO TO 10`	INTEGER(4)	4
`IF (N .EQ. O'123') GO TO 10`	INTEGER(4)	4
`IF (N. EQ. 1HZ) GO TO 10`	INTEGER(4)	4

When a specific data type (generally integer) is required, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`Y(IX) = Y(O'15') + 3.`	INTEGER(4)	4
`Y(IX) = Y(1HA) + 3.`	INTEGER(4)	4

When a nondecimal constant is used as an actual argument, the following occurs:

- For binary, octal, and hexadecimal constants, INTEGER(8) is assumed.
- For Hollerith constants, no data type is assumed.

For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`CALL APAC(Z'34BC2')`	INTEGER(8)	8
`CALL APAC(9HABCDEFGHI)`	None	9

When a binary, octal, or hexadecimal constant is used in any other context, the default integer data type is assumed (default integer can be affected by compiler options). In the following examples, default integer is INTEGER(4):

Statement	Data Type of Constant	Length of Constant (in bytes)
`IF (Z'AF77') 1,2,3`	INTEGER(4)	4
`IF (2HAB) 1,2,3`	INTEGER(4)	4
`I = O'7777' - Z'A39'`[1]	INTEGER(4)	4
`I = 1HC - 1HA`	INTEGER(4)	4
`J = .NOT. O'73777'`	INTEGER(4)	4
`J = .NOT. 1HB`	INTEGER(4)	4

[1]When two typeless constants are used in an operation, they both take default integer type.

When nondecimal constants are not the same length as the length implied by a data type, the following occurs:

- Binary, octal, and hexadecimal constants

These constants can specify up to 16 bytes of data. When the length of the constant is less than the length implied by the data type, the leftmost digits have a value of zero.

When the length of the constant is greater than the length implied by the data type, the constant is truncated on the left. An error results if any nonzero digits are truncated.

Table 15.2 lists the number of bytes that each data type requires.

- Hollerith constants

  When the length of the constant is less than the length implied by the data type, blanks are appended to the constant on the right.

  When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. If any characters other than blank characters are truncated, an error occurs.

  Each Hollerith character occupies one byte of memory.

For More Information:

On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

3.5. Variables

A variable is a data object whose value can be changed at any point in a program. A variable can be any of the following:

- A scalar

  A **scalar** is a single object that has a single value; it can be of any intrinsic or derived (user-defined) type.

- An array

  An **array** is a collection of scalar elements of any intrinsic or derived type. All elements must have the same type and kind parameters.

- A subobject designator

  A subobject is part of an object. The following are subobjects:

  An array element
  An array section
  A structure component
  A character substring

  For example, B(3) is a subobject (array element) designator for array B. A subobject cannot be a variable if its parent object is a constant.

The name of a variable is associated with a single storage location.

Variables are classified by data type, as constants are. The data type of a variable indicates the type of data it contains, including its precision, and implies its storage requirements. When data of any type is assigned to a variable, it is converted to the data type of the variable (if necessary).

A variable is defined when you give it a value. A variable can be defined before program execution by a DATA statement or a type declaration statement. During program execution, variables can be defined or redefined in assignment statements and input statements, or undefined (for example, if an I/O error occurs). When a variable is undefined, its value is unpredictable.

When a variable becomes undefined, all variables associated by storage association also become undefined.

For More Information:

- On arrays, see Section 3.5.2.
- On storage association of variables, see Section 15.5.3.
- On type declaration statements, see Section 5.1.
- On the DATA statement, see Section 5.5.
- On the data type of a numeric expression, see Section 4.1.1.2.

3.5.1. Data Types of Scalar Variables

The data type of a scalar variable can be explicitly declared in a type declaration statement. If no type is declared, the variable has an implicit data type based on predefined typing rules or definitions in an IMPLICIT statement.

An explicit declaration of data type takes precedence over any implicit type. Implicit type specified in an IMPLICIT statement takes precedence over predefined typing rules.

3.5.1.1. Specification of Data Type

Type declaration statements explicitly specify the data type of scalar variables. For example, the following statements associate VAR1 with an 8-byte complex storage location, and VAR2 with an 8-byte double-precision storage location:

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

You can explicitly specify the data type of a scalar variable only once.

If no explicit data type specification appears, any variable with a name that begins with the letter in the range specified in the IMPLICIT statement becomes the data type of the variable.

Character type declaration statements specify that given variables represent character values with the length specified. For example, the following statements associate the variable names INLINE, NAME, and NUMBER with storage locations containing character data of lengths 72, 12, and 9, respectively:

```
CHARACTER*72 INLINE
CHARACTER NAME*12, NUMBER*9
```

In single subprograms, assumed-length character arguments can be used to process character strings with different lengths. The assumed-length character argument has its length specified with an asterisk, for example:

```
CHARACTER*(*) CHARDUMMY
```

The argument CHARDUMMY assumes the length of the actual argument.

For More Information:

- On type declaration statements, see Section 5.1.
- On character type declaration statements, see Section 5.1.2.
- On assumed-length character arguments, see Section 8.8.4.
- On the IMPLICIT statement, see Section 5.9.

3.5.1.2. Implicit Typing Rules

By default, all scalar variables with names beginning with I, J, K, L, M, or N are assumed to be default integer variables. Scalar variables with names beginning with any other letter are assumed to be default real variables. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM_1
TOTAL_NUM	NTOTAL

Names beginning with a dollar sign ($) are implicitly INTEGER.

You can override the default data type implied in a name by specifying data type in either an IMPLICIT statement or a type declaration statement.

For More Information:

- On type declaration statements, see Section 5.1.
- On the IMPLICIT statement, see Section 5.9.

3.5.2. Arrays

An array is a set of scalar elements that have the same type and kind parameters. Any object that is declared with an array specification is an array. Arrays can be declared by using a type declaration statement, or by using a DIMENSION, COMMON, ALLOCATABLE, POINTER, or TARGET statement.

An array can be referenced by element (using subscripts), by section (using a section subscript list), or as a whole. A subscript list (appended to the array name) indicates which array element or array section is being referenced.

A section subscript list consists of subscripts, subscript triplets, or vector subscripts. At least one subscript in the list must be a subscript triplet or vector subscript.

When an array name without any subscripts appears in an intrinsic operation (for example, addition), the operation applies to the whole array (all elements in the array).

An array has the following properties:

- Data type

  An array can have any intrinsic or derived type. The data type of an array (like any other variable) is specified in a type declaration statement or implied by the first letter of its name. All elements of the

array have the same type and kind parameters. If a value assigned to an individual array element is not the same as the type of the array, it is converted to the array's type.

- Rank

  The rank of an array is the number of dimensions in the array. An array can have up to seven dimensions. A rank-one array represents a column of data (a vector), a rank-two array represents a table of data arranged in columns and rows (a matrix), a rank-three array represents a table of data on multiple pages (or planes), and so forth.

- Bounds

  Arrays have a lower and upper bound in each dimension. These bounds determine the range of values that can be used as subscripts for the dimension. The value of either bound can be positive, negative, or zero.

  The bounds of a dimension are defined in an array specification.

- Size

  The size of an array is the total number of elements in the array (the product of the array's extents).

  The **extent** is the total number of elements in a particular dimension. It is determined as follows: upper bound − lower bound + 1.If the value of any of an array's extents is zero, the array has a size of zero.

- Shape

  The shape of an array is determined by its rank and extents, and can be represented as a rank-one array (vector) where each element is the extent of the corresponding dimension.

  Two arrays with the same shape are said to be **conformable**. A scalar is conformable to an array of any shape.

The name and rank of an array must be specified when the array is declared. The extent of each dimension can be constant, but does not need to be. The extents can vary during program execution if the array is a dummy argument array, an automatic array, an array pointer, or an allocatable array.

A whole array is referenced by the array name. Individual elements in a named array are referenced by a scalar subscript or list of scalar subscripts (if there is more than one dimension). A section of a named array is referenced by a section subscript.

Examples

The following are examples of valid array declarations:

```
DIMENSION     A(10, 2, 3)     ! DIMENSION statement
ALLOCATABLE   B(:, :)         ! ALLOCATABLE statement
POINTER       C(:, :, :)      ! POINTER statement
REAL, DIMENSION (2, 5)        ! Type declaration with
                              ! DIMENSION attribute
```

Consider the following array declaration:

```
INTEGER L(2:11,3)
```

The properties of array L are as follows:

Data type:	INTEGER
Rank:	2 (two dimensions)
Bounds:	First dimension: 2 to 11 Second dimension: 1 to 3
Size:	30; the product of the extents: 10 x 3
Shape:	(/10,3/) (or 10 by 3); a vector of the extents 10 and 3

The following example shows other valid ways to declare this array:

```
DIMENSION L(2:11,3)
INTEGER, DIMENSION(2:11,3) :: L
COMMON L(2:11,3)
```

The following example shows references to array elements, array sections, and a whole array:

```
REAL B(10)            ! Declares a rank-one array with 10 elements

INTEGER C(5,8)        ! Declares a rank-two array with 5 elements in
                      !   dimension one and 8 elements in dimension two
...
B(3) = 5.0            ! Reference to an array element
B(2:5) = 1.0          ! Reference to an array section consisting of
                      !   elements: B(2), B(3), B(4), B(5)
...
C(4,8) = I            ! Reference to an array element
C(1:3,3:4) = J        ! Reference to an array section consisting of
                      !   elements:  C(1,3) C(1,4)
                      !              C(2,3) C(2,4)
                      !              C(3,3) C(3,4)
B = 99                ! Reference to a whole array consisting of
                      !   elements: B(1), B(2), B(3), B(4), B(5),
                      !   B(6), B(7), B(8), B(9), and B(10)
```

For More Information:

- On array specifications, see Section 5.1.4.
- On the DIMENSION attribute, see Section 5.6.
- On intrinsic data types, see Section 3.2.
- On derived data types, see Section 3.3.
- On whole arrays, see Section 3.5.2.1.
- On array elements, see Section 3.5.2.2.
- On array sections, see Section 3.5.2.3.
- On intrinsic functions that perform array operations, see Table 9.2.

3.5.2.1. Whole Arrays

A **whole array** is a named array; it is either a named constant or a variable. It is referenced by using the array name (without any subscripts).

If a whole array appears in a nonexecutable statement, the statement applies to the entire array. For example:

```
INTEGER, DIMENSION(2:11,3) :: L   ! Specifies the type and
                                  !    dimensions of array L
```

If a whole array appears in an executable statement, the statement applies to all of the elements in the array. For example:

```
L = 10              ! The value 10 is assigned to all the
                    !   elements in array L

WRITE *, L          ! Prints all the elements in array L
```

3.5.2.2. Array Elements

An **array element** is one of the scalar data items that make up an array. A subscript list (appended to the array or array component) determines which element is being referred to. A reference to an array element takes the following form:

```
array(subscript-list)
```

`array`

Is the name of the array.

`subscript-list`

Is a list of one or more subscripts separated by commas. The number of subscripts must equal the rank of the array.

Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension.

Rules and Behavior

Each array element inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array element cannot inherit the POINTER attribute.

If an array element is of type character, it can be followed by a substring range in parentheses; for example:

```
ARRAY_D(1,2) (1:3)    ! Elements are substrings of length 3
```

However, by convention, such an object is considered to be a substring rather than an array element.

The following are some valid array element references for an array declared as REAL B(10,20): B(1,3), B(10,10), and B(5,8).

For information on forms for array specifications, see Section 5.1.4.

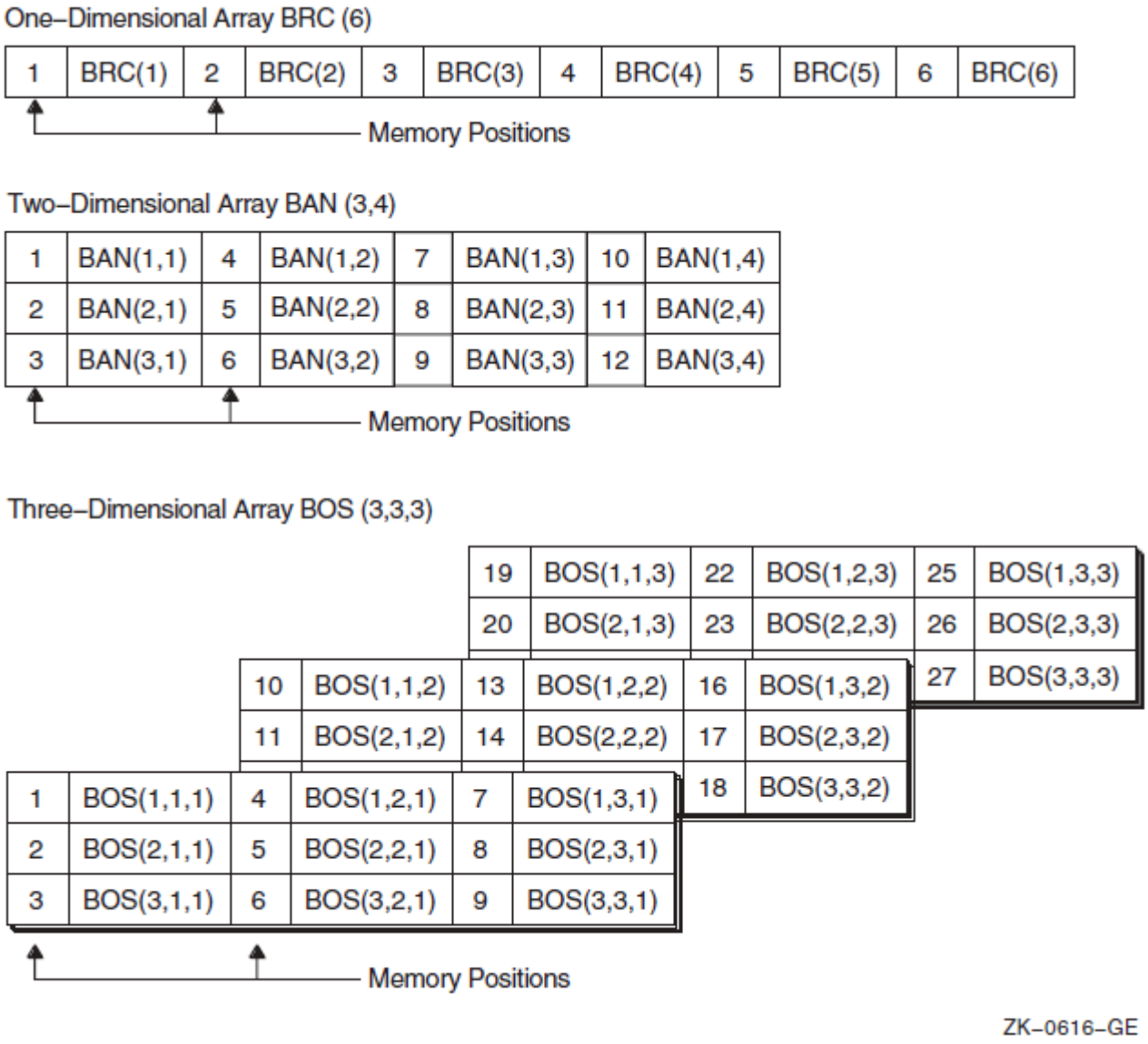
Array Element Order

The elements of an array form a sequence known as array element order. The position of an element in this sequence is its subscript order value.

The elements of an array are stored as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored so that the leftmost subscripts vary most rapidly. This is called the order of subscript progression.

Figure 3.1 shows array storage in one, two, and three dimensions.

Figure 3.1. Array Storage



For example, in two-dimensional array BAN, element BAN(1,2) has a subscript order value of 4; in three-dimensional array BOS, element BOS(1,1,1) has a subscript order value of 1.

In an array section, the subscript order of the elements is their order within the section itself. For example, if an array is declared as B(20), the section B(4:19:4) consists of elements B(4), B(8), B(12), and B(16). The subscript order value of B(4) in the array section is 1; the subscript order value of B(12) in the section is 3.

For More Information

- On substrings, see Section 3.2.5.2.

- On arrays as structure components, see Section 3.3.3.
- On array association, see Section 15.5.3.2.
- On storage sequence association, see Section 15.5.3.

3.5.2.3. Array Sections

An **array section** is a portion of an array that is an array itself. It is an array subobject. A section subscript list (appended to the array or array component) determines which portion is being referred to. A reference to an array section takes the following form:

```
array(sect-subscript-list)
```

`array`

Is the name of the array.

`sect-subscript-list`

Is a list of one or more section subscripts (subscripts, subscript triplets, or vector subscripts) indicating a set of elements along a particular dimension.

At least one of the items in the section subscript list must be a subscript triplet or vector subscript. A subscript triplet specifies array elements in increasing or decreasing order at a given stride. A vector subscript specifies elements in any order.

Each subscript and subscript triplet must be a scalar integer (or other numeric) expression. Each vector subscript must be a rank-one integer expression.

Rules and Behavior

If *no* section subscript list is specified, the rank and shape of the array section is the same as the parent array.

Otherwise, the rank of the array section is the number of vector subscripts and subscript triplets that appear in the list. Its shape is a rank-one array where each element is the number of integer values in the sequence indicated by the corresponding subscript triplet or vector subscript.

If any of these sequences is empty, the array section has a size of zero. The subscript order of the elements of an array section is that of the array object that the array section represents.

Each array section inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array section cannot inherit the POINTER attribute.

If an array (or array component) is of type character, it can be followed by a substring range in parentheses. Consider the following declaration:

```
CHARACTER(LEN=15) C(10,10)
```

In this case, an array section referenced as C(:,:) (1:3) is an array of shape (10,10), whose elements are substrings of length 3 of the corresponding elements of C.

The following shows valid references to array sections. Note that the syntax (/.../) denotes an array constructor (see Section 3.5.2.4).

```
REAL, DIMENSION(20) :: B
...
PRINT *, B(2:20:5)  ! The section consists of elements
                    !     B(2), B(7), B(12), and B(17)

K = (/3, 1, 4/)
B(K) = 0.0      ! Section B(K) is a rank-one array with shape (3) and
                !  size 3. (0.0 is assigned to B(1), B(3), and B(4).)
```

Subscript Triplets

A **subscript triplet** is a set of three values representing the lower bound of the array section, the upper bound of the array section, and the increment (stride) between them. It takes the following form:

```
[first-bound] : [last-bound] [:stride]
```

`first-bound`

Is a scalar integer (or other numeric) expression representing the first value in the subscript sequence. If omitted, the declared lower bound of the dimension is used.

`last-bound`

Is a scalar integer (or other numeric) expression representing the last value in the subscript sequence. If omitted, the declared upper bound of the dimension is used.

When indicating sections of an assumed-size array, this subscript must be specified.

`stride`

Is a scalar integer (or other numeric) expression representing the increment between successive subscripts in the sequence. It must have a nonzero value. If it is omitted, it is assumed to be 1.

The stride has the following effects:

- If the stride is positive, the subscript range starts with the first subscript and is incremented by the value of the stride, until the largest value less than or equal to the second subscript is attained.

  For example, if an array has been declared as B(6,3,2), the array section specified as B(2:4,1:2,2) is a rank-two array with shape (3,2) and size 6. It consists of the following six elements:

  B(2,1,2) B(2,2,2)
  B(3,1,2) B(3,2,2)
  B(4,1,2) B(4,2,2)

  If the first subscript is greater than the second subscript, the range is empty.

- If the stride is negative, the subscript range starts with the value of the first subscript and is decremented by the absolute value of the stride, until the smallest value greater than or equal to the second subscript is attained.

  For example, if an array has been declared as A(15), the array section specified as A(10:3:-2) is a rank-one array with shape (4) and size 4. It consists of the following four elements:

  A(10)
  A(8)
  A(6)

A(4)

If the second subscript is greater than the first subscript, the range is empty.

If a range specified by the stride is empty, the array section has a size of zero.

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used to select the array elements are within the declared bounds. For example, if an array has been declared as A(15), the array section specified as A(4:16:10) is valid. The section is a rank-one array with shape (2) and size 2. It consists of elements A(4) and A(14).

If the subscript triplet does not specify bounds or stride, but only a colon (:), the entire declared range for the dimension is used.

Vector Subscripts

A **vector subscript** is a one-dimensional (rank one) array of integer values (within the declared bounds for the dimension) that selects a section of a whole (parent) array. The elements in the section do not have to be in order and the section can contain duplicate values.

For example, A is a rank-two array of shape (4,6). B and C are rank-one arrays of shape (2) and (3), respectively, with the following values:

```
B = (/1,4/)           ! Syntax (/.../) denotes an array constructor
C = (/2,1,1/)         ! This constructor produces a many-one array section
```

Array section A(3,B) consists of elements A(3,1) and A(3,4). Array section A(C,1) consists of elements A(2,1), A(1,1), and A(1,1). Array section A(B,C) consists of the following elements:

A(1,2) A(1,1) A(1,1)
A(4,2) A(4,1) A(4,1)

An array section with a vector subscript that has two or more elements with the same value is called a **many-one array section**. A many-one section must not appear on the left of the equal sign in an assignment statement, or as an input item in a READ statement.

The following assignments to C also show examples of vector subscripts:

```
INTEGER A(2), B(2), C(2)
...
B    = (/1,2/)
C(B) = A(B)
C    = A((/1,2/))
```

An array section with a vector subscript must not be any of the following:

- An internal file
- An actual argument associated with a dummy array that is defined or redefined (if the INTENT attribute is specified, it must be INTENT(IN))
- The target in a pointer assignment statement

If the sequence specified by the vector subscript is empty, the array section has a size of zero.

For More Information:

- On the INTENT attribute, see Section 5.10.

- On the PARAMETER attribute, see Section 5.14.
- On the TARGET attribute, see Section 5.18.
- On substrings, see Section 3.2.5.2.
- On array sections as structure components, see Section 3.3.3.
- On array constructors, see Section 3.5.2.4.

3.5.2.4. Array Constructors

An **array constructor** can be used to create and assign values to rank-one arrays (and array constants). An array constructor takes the following form:

```
(/ac-value-list/)
```

`ac-value-list`

Is a list of one or more expressions or implied-do loops. Each *ac-value* must have the same type and kind parameters, and be separated by commas.

An implied-do loop in an array constructor takes the following form:

```
(ac-value-expr, do-variable = expr1, expr2 [,expr3])
```

`ac-value-expr`

Is a scalar expression evaluated for each value of the *do-variable* to produce an array element value.

`do-variable`

Is the name of a scalar integer variable. Its scope is that of the implied-do loop.

`expr`

Is a scalar integer expression. The *expr1* and *expr2* specify a range of values for the loop; *expr3* specifies the stride. The *expr3* must be a nonzero value; if it is omitted, it is assumed to be 1.

Rules and Behavior

The array constructed has the same type as the *ac-value-list* expressions.

If the sequence of values specified by the array constructor is empty (there are no expressions or the implied-do loop produces no values), the rank-one array has a size of zero.

An *ac-value* is interpreted as follows:

Form of *ac-value*	**Result**
A scalar expression	Its value is an element of the new array.
An array expression	The values of the elements in the expression (in array element order) are the corresponding sequence of elements in the new array.
An implied-do loop	It is expanded to form a list of array elements under control of the DO variable (like a DO construct).

The following shows the three forms of an *ac-value*:

```
C1 = (/4,8,7,6/)                    ! A scalar expression
C2 = (/B(I, 1:5), B(I:J, 7:9)/)     ! An array expression
C3 = (/(I, I=1, 4)/)                ! An implied-do loop
```

You can also mix these forms, for example:

```
C4 = (/4, A(1:5), (I, I=1, 4), 7/)
```

If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

If the expressions are of type character, Fortran 95/90 requires each expression to have the same character length.

However, VSI Fortran allows the character expressions to be of different character lengths. The length of the resultant character array is the maximum of the lengths of the individual character expressions. For example:

```
print *,len ( (/'a','ab','abc','d'/) )
print *,'++'//(/'a','ab','abc','d'/)//'--'
```

This causes the following to be displayed:

```
           3
 ++a  --++ab --++abc--++d  --
```

If an implied-do loop is contained within another implied-do loop (nested), they cannot have the same DO variable (*do-variable*).

To define arrays of more than one dimension, use the RESHAPE intrinsic function.

The following are alternative forms for array constructors:

- Square brackets (instead of parentheses and slashes) to enclose array constructors; for example, the following two array constructors are equivalent:

  ```
  INTEGER C(4)
  C = (/4,8,7,6/)
  C = [4,8,7,6]
  ```

- A colon-separated triplet (instead of an implied-do loop) to specify a range of values and a stride; for example, the following two array constructors are equivalent:

  ```
  INTEGER D(3)
  D = (/1:5:2/)               ! Triplet form
  D = (/(I, I=1, 5, 2)/)      ! Implied-do loop form
  ```

Examples

The following example shows an array constructor using an implied-do loop:

```
INTEGER ARRAY_C(10)
ARRAY_C = (/(I, I=30, 48, 2)/)
```

The values of ARRAY_C are the even numbers 30 through 48.

The following example shows an array constructor of derived type that uses a structure constructor:

```
TYPE EMPLOYEE
```

```
  INTEGER ID
  CHARACTER(LEN=30) NAME
END TYPE EMPLOYEE

TYPE(EMPLOYEE) CC_4T(4)
CC_4T = (/EMPLOYEE(2732,"JONES"), EMPLOYEE(0217,"LEE"),      &
          EMPLOYEE(1889,"RYAN"), EMPLOYEE(4339,"EMERSON")/)
```

The following example shows how the RESHAPE intrinsic function can be used to create a multidimensional array:

```
E = (/2.3, 4.7, 6.6/)
D = RESHAPE(SOURCE = (/3.5, (/2.0, 1.0/), E/), SHAPE = (/2,3/))
```

D is a rank-two array with shape (2,3) containing the following elements:

3.5 1.0 4.7
2.0 2.3 6.6

For More Information:

- On array element order, see Section 3.5.2.2.
- On the DO construct, see Section 7.6.
- On another way to assign values to arrays, see Section 4.2.1.5.
- On the RESHAPE intrinsic function, see Section 9.4.132.
- On subscript triplets, see Section 3.5.2.3.
- On derived types, see Section 3.3.
- On structure constructors, see Section 3.3.4.
- On array specifications, see Section 5.1.4.

Chapter 4. Expressions and Assignment Statements

4.1. Expressions

An expression represents either a data reference or a computation, and is formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

If the value of an expression is of intrinsic type, it has a kind type parameter. (If the value is of intrinsic type CHARACTER, it also has a length parameter.) If the value of an expression is of derived type, it has no kind type parameter.

An operand is a scalar or array. An operator can be either intrinsic or defined. An intrinsic operator is known to the compiler and is always available to any program unit. A defined operator is described explicitly by a user in a function subprogram and is available to each program unit that uses the subprogram.

The simplest form of an expression (a primary) can be any of the following:

- A constant; for example, 4.2
- A subobject of a constant; for example, 'LMNOP'(2:4)
- A variable; for example, VAR_1
- A structure constructor; for example, EMPLOYEE(3472, "JOHN DOE")
- An array constructor; for example, (/12.0,16.0/)
- A function reference; for example, COS(X)
- Another expression in parentheses; for example, (I+5)

Any variable or function reference used as an operand in an expression must be defined at the time the reference is executed. If the operand is a pointer, it must be associated with a target object that is defined. An integer operand must be defined with an integer value rather than a statement label value. All of the characters in a character data object reference must be defined.

When a reference to an array or an array section is made, all of the selected elements must be defined. When a structure is referenced, all of the components must be defined.

In an expression that has intrinsic operators with an array as an operand, the operation is performed on each element of the array. In expressions with more than one array operand, the arrays must be conformable (they must have the same shape). The operation is applied to corresponding elements of the arrays, and the result is an array of the same shape (the same rank and extents) as the operands.

In an expression that has intrinsic operators with a pointer as an operand, the operation is performed on the value of the target associated with the pointer.

For defined operators, operations on arrays and pointers are determined by the procedure defining the operation.

A scalar is conformable with any array. If one operand of an expression is an array and another operand is a scalar, it is as if the value of the scalar were replicated to form an array of the same shape as the array operand. The result is an array of the same shape as the array operand.

The following sections describe numeric, character, relational, and logical expressions; defined operations; a summary of operator precedence; and initialization and specification expressions.

For More Information:

- On function subprograms that define operators , see Section 8.9.4.
- On arrays, see Section 3.5.2.
- On pointers, see Section 5.15.
- On derived data types, see Section 3.3.

4.1.1. Numeric Expressions

Numeric expressions express numeric computations, and are formed with numeric operands and numeric operators. The evaluation of a numeric operation yields a single numeric value.

The term numeric includes logical data, because logical data is treated as integer data when used in a numeric context. The default for .TRUE. is –1; .FALSE. is 0.

Numeric operators specify computations to be performed on the values of numeric operands. The result is a scalar numeric value or an array whose elements are scalar numeric values. The following are numeric operators:

Operator	**Function**
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus (identity)
–	Subtraction or unary minus (negation)

Unary operators operate on a single operand. **Binary operators** operate on a pair of operands. The plus and minus operators can be unary or binary. When they are unary operators, the plus or minus operators precede a single operand and denote a positive (identity) or negative (negation) value, respectively. The exponentiation, multiplication, and division operators are binary operators.

Valid numeric operations must have results that are defined by the arithmetic used by the processor. For example, raising a negative-valued real to a real power is invalid.

Numeric expressions are evaluated in an order determined by a precedence associated with each operator, as follows (see also Section 4.1.6):

Operator	**Precedence**
**	Highest

Operator	Precedence
* and /	.
Unary + and –	.
Binary + and –	Lowest

Operators with equal precedence are evaluated in left-to-right order. However, exponentiation is evaluated from right to left. For example, A**B**C is evaluated as A**(B**C). B**C is evaluated first, then A is raised to the resulting power.

Normally, two operators cannot appear together. However, VSI Fortran allows two consecutive operators if the second operator is a plus or minus.

Examples

In the following example, the exponentiation operator is evaluated first because it takes precedence over the multiplication operator:

`A**B*C` is evaluated as `(A**B)*C`

Ordinarily, the exponentiation operator would be evaluated first in the following example. However, because VSI Fortran allows the combination of the exponentiation and minus operators, the exponentiation operator is not evaluated until the minus operator is evaluated:

`A**-B*C` is evaluated as `A**(-(B*C))`

Note that the multiplication operator is evaluated first, since it takes precedence over the minus operator.

When consecutive operators are used with constants, the unary plus or minus before the constant is treated the same as any other operator. This can produce unexpected results. In the following example, the multiplication operator is evaluated first, since it takes precedence over the minus operator:

`X/-15.0*Y` is evaluated as `X/-(15.0*Y)`

4.1.1.1. Using Parentheses in Numeric Expressions

You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first. The resulting value is used in the evaluation of the remainder of the expression.

In the following examples, the numbers below the operators indicate a possible order of evaluation. Alternative evaluation orders are possible in the first three examples because they contain operators of equal precedence that are not enclosed in parentheses. In these cases, the compiler is free to evaluate operators of equal precedence in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation.

Expressions within parentheses are evaluated according to the normal order of precedence. In expressions containing nested parentheses, the innermost parentheses are evaluated first.

Nonessential parentheses do not affect expression evaluation, as shown in the following example:

$$4 + 3 * 2 - 6/2 = 7$$
$$\uparrow\ \uparrow\ \uparrow\ \uparrow$$
$$2\ \ 1\ \ 4\ \ 3$$

$$(4 + 3) * 2 - 6/2 = 11$$
$$\uparrow\ \ \uparrow\ \uparrow\ \uparrow$$
$$1\ \ 2\ \ 4\ \ 3$$

$$(4 + 3 * 2 - 6)/2 = 2$$
$$\uparrow\ \uparrow\ \uparrow\ \ \uparrow$$
$$2\ \ 1\ \ 3\ \ 4$$

$$((4 + 3) * 2 - 6)/2 = 4$$
$$\uparrow\ \ \uparrow\ \uparrow\ \ \uparrow$$
$$1\ \ 2\ \ 3\ \ 4$$

However, using parentheses to specify the evaluation order is often important in high-accuracy numerical computations. In such computations, evaluation orders that are algebraically equivalent may not be computationally equivalent when processed by a computer (because of the way intermediate results are rounded off).

Parentheses can be used in argument lists to force a given argument to be treated as an expression, rather than as the address of a memory item.

4.1.1.2. Data Type of Numeric Expressions

If every operand in a numeric expression is of the same data type, the result is also of that type.

If operands of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value depend on the ranking associated with each data type. The following table shows the ranking assigned to each data type:

Data Type	**Ranking**
LOGICAL(1) and BYTE	Lowest
LOGICAL(2)	.
LOGICAL(4)	.
LOGICAL(8)	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(4)	.
INTEGER(8)	.
REAL(4)	.
REAL(8)[1]	.
REAL(16)	.

Data Type	Ranking
COMPLEX(4)	.
COMPLEX(8)[2]	.
COMPLEX(16)	Highest

[1]DOUBLE PRECISION
[2]DOUBLE COMPLEX

The data type of the value produced by an operation on two numeric operands of different data types is the data type of the highest-ranking operand in the operation. For example, the value resulting from an operation on an integer and a real operand is of real type. However, an operation involving a COMPLEX(4) or COMPLEX(8) data type and a DOUBLE PRECISION data type produces a COMPLEX(8) result.

The data type of an expression is the data type of the result of the last operation in that expression, and is determined according to the following conventions:

- Integer operations: Integer operations are performed only on integer operands. Note that logical entities used in a numeric context are treated as integers. In integer arithmetic, any fraction resulting from division is truncated, not rounded. For example, the result of `1/4 + 1/4 + 1/4 + 1/4` is 0, not 1.
- Real operations: Real operations are performed only on real operands or combinations of real, integer, and logical operands. Any integer operands present are converted to real data type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. However, in the statement `Y = (I/J)*X`, an integer division operation is performed on I and J, and a real multiplication is performed on that result and X.

  If any operand is a higher-precision real (REAL(8) or REAL(16)) type, all other operands are converted to that higher-precision real type before the expression is evaluated.

  When a single-precision real operand is converted to a double-precision real operand, low-order binary digits are set to zero. This conversion does not increase accuracy; conversion of a decimal number does not produce a succession of decimal zeros. For example, a REAL variable having the value `0.3333333` is converted to approximately `0.33333331346511840D0`. It is not converted to either `0.3333333000000000D0` or `0.3333333333333333D0`.
- Complex operations: In operations that contain any complex operands, integer operands are converted to real type, as previously described. The resulting single-precision or double-precision operand is designated as the real part of a complex number and the imaginary part is assigned a value of zero. The expression is then evaluated using complex arithmetic and the resulting value is of complex type. Operations involving a COMPLEX(4) or COMPLEX(8) operand and a DOUBLE PRECISION operand are performed as COMPLEX(8) operations; the DOUBLE PRECISION operand is not rounded.

These rules also generally apply to numeric operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a DOUBLE PRECISION (REAL(8)) or REAL(16) representation of the constant were given. For example, the expression`1.0D0 + 0.3333333` is treated as if it is `1.0D0 + 0.3333333000000000D0`.

4.1.2. Character Expressions

A **character expression** consists of a character operator (//) that concatenates two operands of type character. The evaluation of a character expression produces a single value of that type.

The result of a character expression is a character string whose value is the value of the left character operand concatenated to the value of the right operand. The length of a character expression is the sum of the lengths of the values of the operands. For example, the value of the character expression `'AB'//'CDE'` is `'ABCDE'`, which has a length of five.

Parentheses do not affect the evaluation of a character expression; for example, the following character expressions are equivalent:

```
('ABC'//'DE')//'F'
'ABC'//('DE'//'F')
'ABC'//'DE'//'F'
```

Each of these expressions has the value 'ABCDEF'.

If a character operand in a character expression contains blanks, the blanks are included in the value of the character expression. For example, `'ABC '//'D E'//'F '` has a value of `'ABC D EF '`.

4.1.3. Relational Expressions

A **relational expression** consists of two or more expressions whose values are compared to determine whether the relationship stated by the relational operator is satisfied. The following are relational operators:

Operator			**Relationship**
.LT.	or	<	Less than
.LE.	or	<=	Less than or equal to
.EQ.	or	==	Equal to
.NE.	or	/=	Not equal to
.GT.	or	>	Greater than
.GE.	or	>=	Greater than or equal to

The result of the relational expression is .TRUE. if the relation specified by the operator is satisfied; the result is .FALSE. if the relation specified by the operator is not satisfied.

Relational operators are of equal precedence. Numeric operators and the character operator // have a higher precedence than relational operators.

In a numeric relational expression, the operands are numeric expressions. Consider the following example:

```
APPLE+PEACH > PEAR+ORANGE
```

This expression states that the sum of APPLE and PEACH is greater than the sum of PEAR and ORANGE. If this relationship is valid, the value of the expression is .TRUE.; if not, the value is .FALSE..

Operands of type complex can only be compared using the equal operator (== or .EQ.) or the not equal operator (/= or .NE.). Complex entities are equal if their corresponding real and imaginary parts are both equal.

In a character relational expression, the operands are character expressions. In character relational expressions, less than (< or .LT.) means the character value precedes in the ASCII collating sequence, and greater than (> or .GT.) means the character value follows in the ASCII collating sequence. For example:

```
'AB'//'ZZZ' .LT. 'CCCCC'
```

This expression states that 'ABZZZ' is less than 'CCCCC'. In this case, the relation specified by the operator is satisfied, so the result is .TRUE..

Character operands are compared one character at a time, in order, starting with the first character of each operand. If the two character operands are not the same length, the shorter one is padded on the right with blanks until the lengths are equal; for example:

```
'ABC' .EQ. 'ABC  '  'AB' .LT. 'C'
```

The first relational expression has the value .TRUE. even though the lengths of the expressions are not equal, and the second has the value .TRUE. even though 'AB' is longer than 'C'.

A relational expression can compare two numeric expressions of different data types. In this case, the value of the expression with the lower-ranking data type is converted to the higher-ranking data type before the comparison is made

For More Information:

On the ranking of data types, see Section 4.1.1.2.

4.1.4. Logical Expressions

A **logical expression** consists of one or more logical operators and logical, numeric, or relational operands. The following are logical operators:

Operator	Example	Meaning
.AND.	`A .AND. B`	Logical conjunction: the expression is true if both A and B are true.
.OR.	`A .OR. B`	Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.
.NEQV.	`A .NEQV. B`	Logical inequivalence (exclusive OR): the expression is true if either A or B is true, but false if both are true.
.XOR.	`A .XOR. B`	Same as .NEQV.
.EQV.	`A .EQV. B`	Logical equivalence: the expression is true if both A and B are true, or both are false.
.NOT.[1]	`.NOT. A`	Logical negation: the expression is true if A is false and false if A is true.

[1] .NOT. is a unary operator.

Periods cannot appear consecutively except when the second operator is .NOT. For example, the following logical expression is valid:

```
A+B/(A-1) .AND. .NOT. D+B/(D-1)
```

Data Types Resulting from Logical Operations

Logical operations on logical operands produce single logical values (.TRUE. or .FALSE.) of logical type.

Logical operations on integers produce single values of integer type. The operation is carried out bit-by-bit on corresponding bits of the internal (binary) representation of the integer operands.

Logical operations on a combination of integer and logical values also produce single values of integer type. The operation first converts logical values to integers, then operates as it does with integers.

Logical operations cannot be performed on other data types.

Evaluation of Logical Expressions

Logical expressions are evaluated according to the precedence of their operators. Consider the following expression:

```
A*B+C*ABC == X*Y+DM/ZZ .AND. .NOT. K*B > TT
```

This expression is evaluated in the following sequence:

```
(((A*B)+(C*ABC)) == ((X*Y)+(DM/ZZ))) .AND. (.NOT. ((K*B) > TT))
```

As with numeric expressions, you can use parentheses to alter the sequence of evaluation.

When operators have equal precedence, the compiler can evaluate them in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation (except for exponentiation, which is evaluated from right to left).

You should not write logical expressions whose results might depend on the evaluation order of subexpressions. The compiler is free to evaluate subexpressions in any order. In the following example, either (A(I)+1.0) or B(I)*2.0 could be evaluated first:

```
(A(I)+1.0) .GT. B(I)*2.0
```

Some subexpressions might not be evaluated if the compiler can determine the result by testing other subexpressions in the logical expression. Consider the following expression:

```
A .AND. (F(X,Y) .GT. 2.0) .AND. B
```

If the compiler evaluates A first, and A is false, the compiler might determine that the expression is false and might not call the subprogram F(X,Y).

For More Information:

On the precedence of numeric, relational, and logical operators, see Section 4.1.6.

4.1.5. Defined Operations

When operators are defined for functions, the functions can then be referenced as **defined operations**.

The operators are defined by using a generic interface block specifying OPERATOR, followed by the defined operator (in parentheses).

A defined operation is not an intrinsic operation. However, you can use a defined operation to extend the meaning of an intrinsic operator.

For defined unary operations, the function must contain one argument. For defined binary operations, the function must contain two arguments.

Interpretation of the operation is provided by the function that defines the operation.

A Fortran 95/90 defined operator can contain up to 31 letters, and is enclosed in periods (.). Its name cannot be the same name as any of the following:

- The intrinsic operators (.NOT., .AND., .OR., .XOR., .EQV., .NEQV., .EQ., .NE., .GT., .GE., .LT., and .LE.)
- The logical literal constants (.TRUE. or .FALSE.)

An intrinsic operator can be followed by a defined unary operator.

The result of a defined operation can have any type. The type of the result (and its value) must be specified by the defining function.

The following examples show expressions containing defined operators:

```
.COMPLEMENT. A X .PLUS. Y .PLUS. Z M * .MINUS. N
```

For More Information:

- On defining generic operators, see Section 8.9.4.
- On operator precedence, see Section 4.1.6.

4.1.6. Summary of Operator Precedence

Table 4.1 shows the precedence of all intrinsic and defined operators:

Table 4.1. Precedence of Expression Operators

Category	Operator	Precedence
	Defined Unary Operators	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	Unary + or –	.
Numeric	Binary + or –	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE. ==, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.XOR., .EQV., .NEQV.	.
	Defined Binary Operators	Lowest

4.1.7. Initialization and Specification Expressions

A constant expression contains intrinsic operations and parts that are all constants. An initialization expression is a constant expression that is evaluated when a program is compiled. A specification

expression is a scalar, integer expression that is restricted to declarations of array bounds and character lengths.

Initialization and specification expressions can appear in specification statements, with some restrictions.

4.1.7.1. Initialization Expressions

An initialization expression must evaluate at compile time to a constant. It is used to specify an initial value for an entity.

In an initialization expression, each operation is intrinsic and each operand is one of the following:

- A constant or subobject of a constant
- An array constructor where each element and the bounds and strides of each implied-do, are expressions whose primaries are initialization expressions
- A structure constructor whose components are initialization expressions
- An elemental intrinsic function reference of type integer or character, whose arguments are initialization expressions of type integer or character
- A reference to one of the following inquiry functions:

BIT_SIZE	MINEXPONENT
DIGITS	PRECISION
EPSILON	RADIX
HUGE	RANGE
ILEN	SHAPE
KIND	SIZE
LBOUND	TINY
LEN	UBOUND
MAXEXPONENT	

  Each function argument must be one of the following:

  - An initialization expression
  - A variable whose kind type parameter and bounds are not assumed or defined by an ALLOCATE statement, pointer assignment, or an expression that is not an initialization expression

- A reference to one of the following transformational functions (each argument must be an initialization expression):

REPEAT	SELECTED_REAL_KIND
RESHAPE	TRANSFER
SELECTED_INT_KIND	TRIM

- A reference to the transformational function NULL

- An implied-do variable within an array constructor, where the bounds and strides of the corresponding implied-do are initialization expressions
- Another initialization expression enclosed in parentheses

Each subscript, section subscript, and substring starting and ending point must be an initialization expression.

If an initialization expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement (or to the left of the inquiry function in the same statement).

In a specification expression, the number of arguments for a function reference is limited to 255.

Examples

The following examples show valid and invalid initialization (constant) expressions:

Valid	
`-1 + 3`	
`SIZE(B)`	! B is a named constant
`7_2`	
`INT(J, 4)`	! J is a named constant
`SELECTED_INT_KIND (2)`	
Invalid	**Explanation**
`SUM(A)`	Not an allowed function.
`A/4.1 - K**1.2`	Exponential does not have integer power (A and K are named constants).
`HUGE(4.0)`	Argument is not an integer.

For More Information:

- On array constructors, see Section 3.5.2.4.
- On structure constructors, see Section 3.3.4.
- On intrinsic functions, see Chapter 9.

4.1.7.2. Specification Expressions

A specification expression is a restricted expression that is of type integer and has a scalar value. This type of expression appears only in the declaration of array bounds and character lengths.

In a restricted expression, each operation is intrinsic and each operand is one of the following:

- A constant or subobject of a constant
- A variable that is one of the following:
  - A dummy argument that does not have the OPTIONAL or INTENT (OUT) attribute (or the subobject of such a variable)
  - In a common block (or the subobject of such a variable)

  - Made accessible by use or host association (or the subobject of such a variable)
- A structure constructor whose components are restricted expressions
- An implied-do variable within an array constructor, where the bounds and strides of the corresponding implied-do are restricted expressions
- A reference to one of the following inquiry functions:

BIT_SIZE	NWORKERS
DIGITS	PRECISION
EPSILON	PROCESSORS_SHAPE
HUGE	RADIX
ILEN	RANGE
KIND	SHAPE
LBOUND	SIZE
LEN	SIZEOF
MAXEXPONENT	TINY
MINEXPONENT	UBOUND
NUMBER_OF_PROCESSORS	

  Each function argument must be one of the following:

  - A restricted expression
  - A variable whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, are not defined by an expression that is a restricted expression, or are not definable by an ALLOCATE or pointer assignment statement.
- A reference to any other intrinsic function where each argument is a restricted expression.
- A reference to a specification function (see below) where each argument is a restricted expression
- An array constructor where each element and the bounds and strides of each implied-do, are expressions whose primaries are restricted expressions
- Another restricted expression enclosed in parentheses

Each subscript, section subscript, and substring starting and ending point must be a restricted expression.

Specification functions can be used in specification expressions to indicate the attributes of data objects. A specification function is a pure function. It cannot have a dummy procedure argument or be any of the following:

- An intrinsic function
- An internal function
- A statement function
- Defined as RECURSIVE

A variable in a specification expression must have its type and type parameters (if any) specified in one of the following ways:

- By a previous declaration in the same scoping unit
- By the implicit typing rules currently in effect for the scoping unit
- 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 invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement (or to the left of the inquiry function in the same statement).

Examples

The following shows valid specification expressions:

```
MAX(I) + J           ! I and J are scalar integer variables
UBOUND(ARRAY_B,20)   ! ARRAY_B is an assumed-shape dummy array
```

For More Information:

- On array constructors, see Section 3.5.2.4.
- On implicit typing rules, see Section 3.5.1.2.
- On structure constructors, see Section 3.3.4.
- On intrinsic functions, see Chapter 9.
- On use and host association, see Section 15.5.1.2.
- On pure procedures, see Section 8.5.1.2.

4.2. Assignment Statements

An assignment statement causes variables to be defined or redefined. This section describes the following kinds of assignment statements: intrinsic, defined, pointer, masked array (WHERE), and element array (FORALL).

The ASSIGN statement assigns a label to an integer variable. It is discussed in Section 7.2.3.

4.2.1. Intrinsic Assignments

Intrinsic assignment is used to assign a value to a nonpointer variable. In the case of pointers, intrinsic assignment is used to assign a value to the target associated with the pointer variable. The value assigned to the variable (or target) is determined by evaluation of the expression to the right of the equal sign.

An intrinsic assignment statement takes the following form:

```
variable = expression
```

`variable`

Is the name of a scalar or array of intrinsic or derived type (with no defined assignment). The array cannot be an assumed-size array, and neither the scalar nor the array can be declared with the PARAMETER or INTENT(IN) attribute.

`expression`

Is of intrinsic type or the same derived type as *variable*. Its shape must conform with *variable*. If necessary, it is converted to the same type and kind as *variable*.

Rules and Behavior

Before a value is assigned to the variable, the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement.

Note

When the run-time system assigns a value to a scalar integer or character variable and the variable is shorter than the value being assigned, the assigned value may be truncated and significant bits (or characters) lost. This truncation can occur without warning, and can cause the run-time system to pass incorrect information back to the program.

If the variable is a pointer, it must be associated with a definable target. The shape of the target and expression must conform and their type and kind parameters must match.

The following sections discuss numeric, logical, character, derived-type, and array intrinsic assignment.

For More Information:

- On subroutine subprograms that define assignment, see Section 8.9.5.
- On arrays, see Section 3.5.2.
- On pointers, see Section 5.15.
- On derived data types, see Section 3.3.

4.2.1.1. Numeric Assignment Statements

For numeric assignment statements, the variable and expression must be numeric type.

The expression must yield a value that conforms to the range requirements of the variable. For example, a real expression that produces a value greater than 32767 is invalid if the entity on the left of the equal sign is an INTEGER(2) variable.

Significance can be lost if an INTEGER(4) value, which can exactly represent values of approximately the range –2*10**9 to +2*10**9, is converted to REAL(4) (including the real part of a complex constant), which is accurate to only about seven digits.

If the variable has the same data type as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the variable before it is assigned.

Table 4.2 summarizes the data conversion rules for numeric assignment statements.

Table 4.2. Conversion Rules for Numeric Assignment Statements

Scalar Memory Reference (V)	**Expression (E)**	
	Integer, Logical or Real	**Complex**
Integer or Logical	V=INT(E)	V=INT(REAL(E)) Imaginary part of E is not used.
REAL (KIND=4)	V=REAL(E)	V=REAL(REAL(E)) Imaginary part of E is not used.
REAL (KIND=8)	V=DBLE(E)	V=DBLE(REAL(E)) Imaginary part of E is not used.
REAL (KIND=16)	V=QEXT(E)	V=QEXT(REAL(E)) Imaginary part of E is not used.
COMPLEX (KIND=4)	V=CMPLX(REAL(E), 0.0)	V=CMPLX(REAL(REAL(E)), REAL(AIMAG(E)))
COMPLEX (KIND=8)	V=CMPLX(DBLE(E), 0.0)	V=CMPLX(DBLE(REAL(E)), DBLE(AIMAG(E)))
COMPLEX (KIND=16)	V=CMPLX(QEXT(E), 0.0)	V=CMPLX(QEXT(REAL(E)), QEXT(AIMAG(E)))

For more information on the referenced intrinsic functions, see Chapter 9.

Examples

The following examples demonstrate valid and invalid numeric assignment statements:

Valid

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))
PI = 3.14159
SUM = SUM + 1.
ARRAY_A = ARRAY_B +        ! Valid if all arrays conform in ! shape
 ARRAY_C + SCALAR_I
```

Invalid	**Explanation**
`3.14 = A - B`	Entity on the left must be a variable.
`ICOUNT = A//B(3:7)`	Implicitly typed data types do not match.
`SCALAR_I = ARRAY_A(:)`	Shapes do not match.

4.2.1.2. Logical Assignment Statements

For logical assignment statements, the variable must be of logical type and the expression can be of logical or numeric type.

If necessary, the expression is converted to the same type and kind as the variable.

Examples

The following examples demonstrate valid logical assignment statements:

```
PAGEND = .FALSE.
```

```
PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND
ABIG = A.GT.B .AND. A.GT.C .AND. A.GT.D
LOGICAL_VAR = 123   ! Moves binary value of 123 to LOGICAL_VAR
```

4.2.1.3. Character Assignment Statements

For character assignment statements, the variable and expression must be of character type and have the same kind parameter.

The variable and expression can have different lengths. If the length of the expression is greater than the length of the variable, the character expression is truncated on the right. If the length of the expression is less than the length of the variable, the character expression is filled on the right with blank characters.

If you assign a value to a character substring, you do not affect character positions in any part of the character scalar variable not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged. If the character position is undefined, it remains undefined.

Examples

The following examples demonstrate valid and invalid character assignment statements. (In the valid examples, all variables are of type character).

Valid	
`FILE = 'PROG2'`	
`REVOL(1) = 'MAR'//'CIA'`	
`LOCA(3:8) = 'PLANT5'`	
`TEXT(I,J+1)(2:N-1) = NAME//X`	
Invalid	**Explanation**
`'ABC' = CHARS`	Left element must be a character variable, array element, or substring reference.
`CHARS = 25`	Expression does not have a character data type.
`STRING = 5HBEGIN`	Expression does not have a character data type. Note that Hollerith constants are numeric, not character.

4.2.1.4. Derived-Type Assignment Statements

In derived-type assignment statements, the variable and expression must be of the same derived type. There must be no accessible interface block with defined assignment for objects of this derived type.

The derived-type assignment is performed as if each component of the expression is assigned to the corresponding component of the variable. Pointer assignment is performed for pointer components, and intrinsic assignment is performed for nonpointer components.

Examples

The following example demonstrates derived-type assignment:

```
TYPE DATE
  LOGICAL(1) DAY, MONTH
  INTEGER(2) YEAR
END TYPE DATE
```

```
TYPE(DATE) TODAY, THIS_WEEK(7)

TYPE APPOINTMENT
...
  TYPE(DATE) APP_DATE
END TYPE

TYPE(APPOINTMENT) MEETING

DO I = 1,7
  CALL GET_DATE(TODAY)
  THIS_WEEK(I) = TODAY
END DO
MEETING%APP_DATE = TODAY
```

For More Information:

- On derived types, see Section 3.3.
- On pointer assignment, see Section 4.2.3.

4.2.1.5. Array Assignment Statements

Array assignment is permitted when the array expression on the right has the same shape as the array variable on the left, or the expression on the right is a scalar.

If the expression is a scalar, and the variable is an array, the scalar value is assigned to every element of the array.

If the expression is an array, the variable must also be an array. The array element values of the expression are assigned (element by element) to corresponding elements of the array variable.

A **many-one array section** is a vector-valued subscript that has two or more elements with the same value. In intrinsic assignment, the variable cannot be a many-one array section because the result of the assignment is undefined.

Examples

In the following example, X and Y are arrays of the same shape:

```
X = Y
```

The corresponding elements of Y are assigned to those of X element by element; the first element of Y is assigned to the first element of X, and so forth. The processor can perform the element-by-element assignment in any order.

The following example shows a scalar assigned to an array:

```
B(C+1:N, C) = 0
```

This sets the elements B (C+1,C), B (C+2,C),...B (N,C) to zero.

The following example causes the values of the elements of array A to be reversed:

```
REAL A(20)
```

```
...
A(1:20) = A(20:1:-1)
```

For More Information:

- On arrays, see Section 3.5.2.
- On masked array assignment, see Section 4.2.4.
- On element array assignment, see Section 4.2.5.
- On array constructors, see Section 3.5.2.4.

4.2.2. Defined Assignments

Defined assignment specifies an assignment operation. It is defined by a subroutine subprogram containing a generic interface block with the specifier ASSIGNMENT(=). The subroutine is specified by a SUBROUTINE or ENTRY statement that has two nonoptional dummy arguments.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

The dummy arguments represent the variable and expression, in that order. The rank (and shape, if either or both are arrays), type, and kind parameters of the variable and expression in the assignment statement must match those of the corresponding dummy arguments.

The dummy arguments must not both be numeric, or of type logical or character with the same kind parameter.

If the variable in an elemental assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of the variable and expression. If the expression is scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of the expression.

For More Information:

- On subroutine subprograms, see Section 8.5.3.
- On subroutine subprograms that define assignment, see Section 8.9.5.
- On derived data types, see Section 3.3.
- On intrinsic operations, see Sections 4.1.1 and 4.1.2.

4.2.3. Pointer Assignments

In ordinary assignment involving pointers, the pointer is an alias for its target. In pointer assignment, the pointer is associated with a target. If the target is undefined or disassociated, the pointer acquires the same status as the target. The pointer assignment statement has the following form:

```
pointer-object => target
```

`pointer-object`

Is a variable name or structure component declared with the POINTER attribute.

`target`

Is a variable or expression. Its type and kind parameters, and rank must be the same as *pointer-object*. It cannot be an array section with a vector subscript.

Rules and Behavior

If the target is a variable, it must have the POINTER or TARGET attribute, or be a subobject whose parent object has the TARGET attribute.

If the target is an expression, the result must be a pointer.

If the target is not a pointer (it has the TARGET attribute), the pointer object is associated with the target.

If the target is a pointer (it has the POINTER attribute), its status determines the status of the pointer object, as follows:

- If the pointer is associated, the pointer object is associated with the same object as the target
- If the pointer is disassociated, the pointer object becomes disassociated
- If the pointer is undefined, the pointer object becomes undefined

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined.

When pointer assignment occurs, any previous association between the pointer object and a target is terminated.

Pointers can also be assigned for a pointer structure component by execution of a derived-type intrinsic assignment statement or a defined assignment statement.

Pointers can also become associated by using the ALLOCATE statement to allocate the pointer.

Pointers can become disassociated by deallocation, nullification of the pointer (using the DEALLOCATE or NULLIFY statements), or by reference to the NULL intrinsic function.

Examples

The following are examples of pointer assignments:

```
HOUR => MINUTES(1:60)          ! target is an array
M_YEAR => MY_CAR%YEAR          ! target is a structure component
NEW_ROW%RIGHT => CURRENT_ROW   ! pointer object is a structure component
PTR => M                       ! target is a variable
POINTER_C => NULL ()           ! reference to NULL intrinsic
```

The following example shows a target as a pointer:

```
INTEGER, POINTER :: P, N
INTEGER, TARGET :: M
INTEGER S
M = 14
N => M                          ! N is associated with M
P => N                          ! P is associated with M through N
S = P + 5
```

The value assigned to S is 19 (14 + 5).

For More Information:

- On arrays, see Section 3.5.2.
- On pointers, see Section 5.15.
- On the ALLOCATE, DEALLOCATE, and NULLIFY statements, see Chapter 6.
- On derived-type intrinsic assignments, see Section 4.2.1.
- On defined assignment, see Section 4.2.2.
- On the NULL intrinsic function, see Section 9.4.110.

4.2.4. WHERE Statement and Construct

The WHERE statement and construct let you use masked array assignment, which performs an array operation on selected elements. This kind of assignment applies a logical test to an array on an element-by-element basis.

The WHERE statement takes the following form:

```
WHERE (mask-expr1) assign-stmt
```

The WHERE construct takes the following form:

```
[name:] WHERE (mask-expr1)
   [where-body-stmt]...
[ELSEWHERE (mask-expr2) [name:]
   [where-body-stmt]...]
[ELSEWHERE [name:]   [where-body-stmt]...]
END WHERE [name:]
```

`mask-expr1, mask-expr2`

Are logical array expressions (called mask expressions).

`assign-stmt`

Is an assignment statement of the form: array variable = array expression.

`name`

Is the name of the WHERE construct.

`where-body-stmt`

Is one of the following:

- An *assign-stmt*

  This can be a defined assignment only if the routine implementing the defined assignment is elemental.

- A WHERE statement or construct

Rules and Behavior

If a construct name is specified in a WHERE statement, the same name must appear in the corresponding END WHERE statement. The same construct name can optionally appear in any ELSEWHERE statement in the construct. (ELSEWHERE cannot specify a different name).

In each assignment statement, the mask expression, the variable being assigned to, and the expression on the right side, must all be conformable. Also, the assignment statement cannot be a defined assignment.

Only the WHERE statement (or the first line of the WHERE construct) can be labeled as a branch target statement.

The following is an example of a WHERE statement:

```
INTEGER A, B, C
DIMENSION A(5), B(5), C(5)
DATA A /0,1,1,1,0/
DATA B /10,11,12,13,14/
C = -1

WHERE(A .NE. 0) C = B / A
```

The resulting array C contains: –1,11,12,13, and –1.

The assignment statement is only executed for those elements where the mask is true. Think of the mask expression as being evaluated first into a logical array that has the value true for those elements where A is positive. This array of trues and falses is applied to the arrays A, B and C in the assignment statement. The right side is only evaluated for elements for which the mask is true; assignment on the left side is only performed for those elements for which the mask is true. The elements for which the mask is false do not get assigned a value.

In a WHERE construct, the mask expression is evaluated first and only once. Every assignment statement following the WHERE is executed as if it were a WHERE statement with " *mask-expr1*" and every assignment statement following the ELSEWHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSEWHERE specifies " *mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSEWHERE statement.

You should be careful if the statements have side effects, or modify each other or the mask expression.

The following is an example of the WHERE construct:

```
DIMENSION PRESSURE(1000), TEMP(1000), PRECIPITATION(1000)
WHERE(PRESSURE .GE. 1.0)
  PRESSURE = PRESSURE + 1.0
  TEMP = TEMP - 10.0
ELSEWHERE
  PRECIPITATION = .TRUE.
ENDWHERE
```

The mask is applied to the arguments of functions on the right side of the assignment if they are considered to be elemental functions. Only elemental intrinsics are considered elemental functions. Transformational intrinsics, inquiry intrinsics, and functions or operations defined in the subprogram are considered to be nonelemental functions.

Consider the following example using LOG, an elemental function:

```
WHERE(A .GT. 0)  B = LOG(A)
```

The mask is applied to A, and LOG is executed only for the positive values of A. The result of the LOG is assigned to those elements of B where the mask is true.

Consider the following example using SUM, a nonelemental function:

```
REAL A, B
DIMENSION A(10,10), B(10)
WHERE(B .GT. 0.0)  B = SUM(A, DIM=1)
```

Since SUM is nonelemental, it is evaluated fully for all of A. Then, the assignment only happens for those elements for which the mask evaluated to true.

Consider the following example:

```
REAL A, B, C
DIMENSION A(10,10), B(10), C(10)
WHERE(C .GT. 0.0)  B = SUM(LOG(A), DIM=1)/C
```

Because SUM is nonelemental, all of its arguments are evaluated fully regardless of whether they are elemental or not. In this example, LOG(A) is fully evaluated for all elements in A even though LOG is elemental. Notice that the mask is applied to the result of the SUM and to C to determine the right side. One way of thinking about this is that everything inside the argument list of a nonelemental function does not use the mask, everything outside does.

For More Information:

On a generalized form of masked array assignment, see Section 4.2.5.

4.2.5. FORALL Statement and Construct

The FORALL statement and construct is a generalization of the Fortran 95/90 masked array assignment (WHERE statement and construct). It allows more general array shapes to be assigned, especially in construct form.

FORALL is a feature of Fortran 95. It takes the following form:

```
FORALL (triplet-spec [,triplet-spec]...[,mask-expr]) assign-stmt
```

The FORALL construct takes the following form:

```
[name:] FORALL (triplet-spec [,triplet-spec]...[,mask-expr])
   forall-body-stmt
   [forall-body-stmt]...
END FORALL [name]
```

`triplet-spec`

Is a triplet specification with the following form:

```
subscript-name = subscript-1 : subscript-2 [:stride]
```

The *subscript-name* must be a scalar of type integer. It is valid only within the scope of the FORALL; its value is undefined on completion of the FORALL.

The *subscript*s and *stride* cannot contain a reference to any *subscript-name* in *triplet-spec*.

The *stride* cannot be zero. If it is omitted, the default value is 1.

Evaluation of an expression in a triplet specification must not affect the result of evaluating any other expression in another triplet specification.

`mask-expr`

Is a logical array expression (called the mask expression). If it is omitted, the value .TRUE. is assumed. The mask expression can reference the subscript name in *triplet-spec*.

`assign-stmt`

Is an assignment statement or a pointer assignment statement. The variable being assigned to must be an array element or array section and must reference all subscript names included in all *triplet-spec*s.

`name`

Is the name of the FORALL construct.

`forall-body-stmt`

Is one of the following:

- An *assignment-stmt*
- A WHERE statement or construct

  The WHERE statement and construct use a mask to make the array assignments (see Section 4.2.4).
- A FORALL statement or construct

Rules and Behavior

If a construct name is specified in the FORALL statement, the same name must appear in the corresponding END FORALL statement.

A FORALL statement is executed by first evaluating all bounds and stride expressions in the triplet specifications, giving a set of values for each subscript name. The FORALL assignment statement is executed for all combinations of subscript name values for which the mask expression is true.

The FORALL assignment statement is executed as if all expressions (on both sides of the assignment) are completely evaluated before any part of the left side is changed. Valid values are assigned to corresponding elements of the array being assigned to. No element of an array can be assigned a value more than once.

A FORALL construct is executed as if it were multiple FORALL statements, with the same triplet specifications and mask expressions. Each statement in the FORALL body is executed completely before execution begins on the next FORALL body statement.

Any procedure referenced in the mask expression or FORALL assignment statement must be pure.

Pure functions can be used in the mask expression or called directly in a FORALL statement. Pure subroutines cannot be called directly in a FORALL statement, but can be called from other pure procedures.

Examples

Consider the following:

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0) B(I, J) = 1.0 / A(I, J)
```

This statement takes the reciprocal of each nonzero element of array A(1:N, 1:N) and assigns it to the corresponding element of array B. Elements of A that are zero do not have their reciprocal taken, and no assignments are made to corresponding elements of B.

Every array assignment statement and WHERE statement can be written as a FORALL statement, but some FORALL statements cannot be written using just array syntax. For example, the preceding FORALL statement is equivalent to the following:

```
WHERE(A /= 0.0) B = 1.0 / A
```

It is also equivalent to:

```
FORALL (I = 1:N, J = 1:N)
  WHERE(A(I, J) .NE. 0.0) B(I, J) = 1.0/A(I, J)
END FORALL
```

However, the following FORALL example cannot be written using just array syntax:

```
FORALL(I = 1:N, J = 1:N) H(I, J) = 1.0/REAL(I + J - 1)
```

This statement sets array element H(I, J) to the value `1.0/REAL(I + J - 1)` for values of I and J between 1 and N.

Consider the following:

```
TYPE MONARCH
  INTEGER, POINTER :: P
END TYPE MONARCH

TYPE(MONARCH), DIMENSION(8)   :: PATTERN
INTEGER, DIMENSION(8), TARGET :: OBJECT
FORALL(J=1:8)  PATTERN(J)%P => OBJECT(1+IEOR(J-1,2))
```

This FORALL statement causes elements 1 through 8 of array PATTERN to point to elements 3, 4, 1, 2, 7, 8, 5, and 6, respectively, of OBJECT. IEOR can be referenced here because it is pure.

The following example shows a FORALL construct:

```
FORALL(I = 3:N + 1, J = 3:N + 1)
  C(I, J) = C(I, J + 2) + C(I, J - 2) + C(I + 2, J) + C(I - 2, J)
  D(I, J) = C(I, J)
END FORALL
```

The assignment to array D uses the values of C computed in the first statement in the construct, not the values before the construct began execution.

For More Information:

- On subscript triplets, see Section 3.5.2.3.
- On pointer assignment, see Section 4.2.3.
- On the WHERE statement and construct, see Section 4.2.4.
- On pure procedures, see Section 8.5.1.2.

- On the FORALL statement and construct, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Chapter 5. Specification Statements

A **specification statement** is a nonexecutable statement that declares the attributes of data objects. In Fortran 95/90, many of the attributes that can be defined in specification statements can also be optionally specified in type declaration statements.

This chapter contains information on the following topics:

- Type declaration statement (Section 5.1)

  Explicitly specifies the properties (for example: data type, rank, and extent) of data objects.

- ALLOCATABLE attribute and statement (Section 5.2)

  Specifies a list of array names that are allocatable (have a deferred-shape).

- AUTOMATIC and STATIC attributes and statements (Section 5.3)

  Control the storage allocation of variables in subprograms.

- COMMON statement (Section 5.4)

  Defines one or more contiguous areas, or blocks, of physical storage (called common blocks).

- DATA statement (Section 5.5)

  Assigns initial values to variables before program execution.

- DIMENSION attribute and statement (Section 5.6)

  Specifies that an object is an array, and defines the shape of the array.

- EQUIVALENCE statement (Section 5.7)

  Specifies that a storage area is shared by two or more objects in a program unit.

- EXTERNAL attribute and statement (Section 5.8)

  Allows external (user-supplied) procedures to be used as arguments to other subprograms.

- IMPLICIT statement (Section 5.9)

  Overrides the implicit data type of names.

- INTENT attribute and statement (Section 5.10)

  Specifies the intended use of a dummy argument.

- INTRINSIC attribute and statement (Section 5.11)

  Allows intrinsic procedures to be used as arguments to subprograms.

- NAMELIST statement (Section 5.12)

  Associates a name with a list of variables. This group name can be referenced in some input/output operations.

- OPTIONAL attribute and statement (Section 5.13)

  Allows a procedure reference to omit arguments.

- PARAMETER attribute and statement (Section 5.14)

  Defines a named constant.

- POINTER attribute and statement (Section 5.15)

  Specifies that an object is a pointer.

- PRIVATE and PUBLIC attributes and statements (Section 5.16)

  Declare the accessibility of entities in a module.

- SAVE attribute and statement (Section 5.17)

  Causes the definition and status of objects to be retained after the subprogram in which they are declared completes execution.

- TARGET attribute and statement (Section 5.18)

  Specifies a pointer target.

- VOLATILE attribute and statement (Section 5.19)

  Prevents optimizations from being performed on specified objects.

For more information on BLOCK DATA and PROGRAM statements, see Chapter 8.

5.1. Type Declaration Statements

A type declaration statement explicitly specifies the properties of data objects or functions.

The general form of a type declaration statement follows:

```
type [[,att]... ::] v [/c-list/] [,v [/c-list/]]...
```

`type`

Is one of the following data type specifiers:

BYTE	DOUBLE COMPLEX
INTEGER[([KIND=]k)]	CHARACTER[([LEN=]n)[,[KIND=]k]]
REAL[([KIND=]k)]	LOGICAL[([KIND=]k)]
DOUBLE PRECISION	TYPE (derived-type-name)
COMPLEX[([KIND=]k)]	

In the optional kind selector "([KIND=]k)", *k* is the kind parameter. It must be an acceptable kind parameter for that data type. If the kind selector is not present, entities declared are of default type. (For a list of the valid noncharacter data types, see Table 5.2).

Kind parameters for intrinsic numeric and logical data types can also be specified using the *n format, where n is the length (in bytes) of the entity; for example, INTEGER*4.

att

Is one of the following attribute specifiers:

ALLOCATABLE	(Section 5.2)	POINTER	(Section 5.15)
AUTOMATIC	(Section 5.3)	PRIVATE[1]	(Section 5.16)
DIMENSION	(Section 5.6)	PUBLIC [1]	Section 5.16)
EXTERNAL	(Section 5.8)	SAVE	(Section 5.17)
INTENT	(Section 5.10)	STATIC	(Section 5.3)
INTRINSIC	(Section 5.11)	TARGET	(Section 5.18)
OPTIONAL	(Section 5.13)	VOLATILE	(Section 5.19)
PARAMETER	(Section 5.14)		

[1]These are access specifiers.

v

Is the name of a data object or function. It can optionally be followed by:

- An array specification, if the object is an array.

  In a function declaration, an array must be a deferred-shape array if it has the POINTER attribute; otherwise, it must be an explicit-shape array.

- A character length, if the object is of type character.

- An initialization expression or, for pointer objects, => NULL().

A function name must be the name of an intrinsic function, external function, function dummy procedure, or statement function.

c-list

Is a list of constants, as in a DATA statement. If v is the name of a constant or an initialization expression, the c-list cannot be present.

The c-list cannot specify more than one value unless it initializes an array. When initializing an array, the c-list must contain a value for every element in the array.

Rules and Behavior

Type declaration statements must precede all executable statements.

In most cases, a type declaration statement overrides (or confirms) the implicit type of an entity. However, a variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The double colon separator (::) is required only if the declaration contains an attribute specifier or initialization; otherwise it is optional.

If att appears, c-list cannot be specified. For example:

```
INTEGER I /2/              ! Valid
```

```
INTEGER, SAVE :: I /2/    ! Invalid
```

The same attribute must not appear more than once in a given type declaration statement, and an entity cannot be given the same attribute more than once in a scoping unit.

If the PARAMETER attribute is specified, the declaration must contain an initialization expression.

If => NULL() is specified for a pointer, its initial association status is disassociated.

A variable (or variable subobject) can only be initialized once in an executable program.

If a declaration contains an initialization expression, but no PARAMETER attribute is specified, the object is a variable whose value is initially defined. The object becomes defined with the value determined from the initialization expression according to the rules of intrinsic assignment.

The presence of initialization implies that the name of the object is saved, except for objects in named common blocks or objects with the PARAMETER attribute.

The following objects cannot be initialized in a type declaration statement:

- Dummy argument
- Function result
- Object in a named common block (unless the type declaration is in a block data program unit)
- Object in blank common
- Allocatable array
- External name
- Intrinsic name
- Automatic object
- Object that has the AUTOMATIC attribute

An object can have more than one attribute. Table 5.1 shows compatible attributes.

Table 5.1. Compatible Attributes

Attribute	**Compatible with:**
ALLOCATABLE	AUTOMATIC, DIMENSION[1], PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
AUTOMATIC	ALLOCATABLE, DIMENSION, POINTER, TARGET, VOLATILE
DIMENSION	ALLOCATABLE, AUTOMATIC, INTENT, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
EXTERNAL	OPTIONAL, PRIVATE, PUBLIC
INTENT	DIMENSION, OPTIONAL, TARGET, VOLATILE
INTRINSIC	PRIVATE, PUBLIC
OPTIONAL	DIMENSION, EXTERNAL, INTENT, POINTER, TARGET, VOLATILE

Attribute	Compatible with:
PARAMETER	DIMENSION, PRIVATE, PUBLIC
POINTER	AUTOMATIC, DIMENSION [1], OPTIONAL, PRIVATE, PUBLIC, SAVE, STATIC, VOLATILE
PRIVATE	ALLOCATABLE, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, TARGET, VOLATILE
PUBLIC	ALLOCATABLE, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, TARGET, VOLATILE
SAVE	ALLOCATABLE, DIMENSION, POINTER, PRIVATE, PUBLIC, STATIC, TARGET, VOLATILE
STATIC	ALLOCATABLE, DIMENSION, POINTER, PRIVATE, PUBLIC, SAVE, TARGET, VOLATILE
TARGET	ALLOCATABLE, AUTOMATIC, DIMENSION, INTENT, OPTIONAL, PRIVATE, PUBLIC, SAVE, STATIC, VOLATILE
VOLATILE	ALLOCATABLE, AUTOMATIC, DIMENSION, INTENT, OPTIONAL, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET

[1]With deferred shape.

Examples

The following show valid type declaration statements:

```
DOUBLE PRECISION B(6)
INTEGER(KIND=2) I
REAL(KIND=4) X, Y
REAL(4) X, Y
LOGICAL, DIMENSION(10,10) :: ARRAY_A, ARRAY_B
INTEGER, PARAMETER :: SMALLEST = SELECTED_REAL_KIND(6, 70)
REAL(KIND (0.0)) M
COMPLEX(KIND=8) :: D
TYPE(EMPLOYEE) :: MANAGER
REAL, INTRINSIC :: COS
CHARACTER(15) PROMPT
CHARACTER*12, SAVE :: HELLO_MSG
INTEGER COUNT, MATRIX(4,4), SUM
LOGICAL*2 SWITCH
REAL :: X = 2.0
TYPE (NUM), POINTER :: FIRST => NULL()
```

For More Information:

- On specific kind parameters of intrinsic data types, see Section 3.2.
- On derived data types, see Section 3.3.
- On implicit typing, see Section 3.5.1.2.
- On the DATA statement, see Section 5.5.
- On initialization expressions, see Section 4.1.7.1.

5.1.1. Declaration Statements for Noncharacter Types

Table 5.2 shows the data types that can appear in noncharacter type declaration statements.

Table 5.2. Noncharacter Data Types

BYTE[1]
LOGICAL[2]
LOGICAL(1) (or LOGICAL*1)
LOGICAL(2) (or LOGICAL*2)
LOGICAL(4) (or LOGICAL*4)
LOGICAL(8) (or LOGICAL*8)
INTEGER[3]
INTEGER(1) (or INTEGER*1)
INTEGER(2) (or INTEGER*2)
INTEGER(4) (or INTEGER*4)
INTEGER(8) (or INTEGER*8)
REAL[4]
REAL(4) (or REAL*4)
DOUBLE PRECISION (REAL(8)) (or REAL*8)
REAL(16) (or REAL*16)
COMPLEX[5]
COMPLEX(4) (or COMPLEX*8)
DOUBLE COMPLEX (COMPLEX(8)) (or COMPLEX*16)
COMPLEX(16) (or COMPLEX*32)

[1]Same as INTEGER(1).

[2]This is treated as default logical.

[3]This is treated as default integer.

[4]This is treated as default real.

[5]This is treated as default complex.

In noncharacter type declaration statements, you can optionally specify the name of the data object or function as v*n, where n is the length (in bytes) of v. The length specified overrides the length implied by the data type.

The value for n must be a valid length for the type of v (see Table 15.2). The type specifiers BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX have one valid length, so the n specifier is invalid for them.

For an array specification, the n must be placed immediately following the array name; for example, in an INTEGER declaration statement, IVEC*2(10) is an INTEGER(2) array of 10 elements.

Examples

In a noncharacter type declaration statement, a subsequent kind parameter overrides any initial kind parameter. For example, consider the following statements:

```
INTEGER(2) I, J, K, M12*4, Q, IVEC*4(10)
REAL(8) WX1, WXZ, WX3*4, WX5, WX6*4
```

```
REAL(8) PI/3.14159E0/, E/2.72E0/, QARRAY(10)/5*0.0,5*1.0/
```

In the first statement, M12*4 and IVEC*4 override the KIND=2 specification. In the second statement, WX3*4 and WX6*4 override the KIND=8 specification. In the third statement, QARRAY is initialized with implicit conversion of the REAL(4) constants to a REAL(8) data type.

For More Information:

- On compiler options that can affect the defaults for numeric and logical data types, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the general form and rules for type declaration statements, see Section 5.1.

5.1.2. Declaration Statements for Character Types

A CHARACTER type specifier can be immediately followed by the length of the character object or function. It takes one of the following forms:

```
CHARACTER [([LEN=]len)]
CHARACTER [([LEN=]len [,[KIND=]n])]
CHARACTER [(KIND=n [,LEN=len])]

CHARACTER*len[,]
```

len

Is one of the following:

- In keyword forms

  The *len* is a specification expression or an asterisk (*). If no length is specified, the default length is 1.

  If the length evaluates to a negative value, the length of the character entity is zero.

- In nonkeyword form

  The *len* is a specification expression or an asterisk enclosed in parentheses, or a scalar integer literal constant (with no kind parameter). The comma is permitted only if no double colon (::) appears in the type declaration statement.

  This form can also (optionally) be specified following the name of the data object or function (*v*len*). In this case, the length specified overrides any length following the CHARACTER type specifier.

The largest valid value for *len* in both forms is 65535. Negative values are treated as zero.

n

Is a scalar integer initialization expression specifying a valid kind parameter. Currently the only kind available is 1.

Rules and Behavior

An automatic object can appear in a character declaration. The object cannot be a dummy argument, and its length must be declared with a specification expression that is not a constant expression.

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

When an asterisk length specification * (*) is used for a function name or dummy argument, it assumes the length of the corresponding function reference or actual argument. Similarly, when an asterisk length specification is used for a named constant, the name assumes the length of the actual constant it represents. For example, STRING assumes a 9-byte length in the following statements:

```
CHARACTER*(*) STRING
PARAMETER (STRING = 'VALUE IS:')
```

A function name must not be declared with a * length if the function is an internal or module function, or if it is array-valued, pointer-valued, recursive, or pure.

The form CHARACTER*(*) is an obsolescent feature in Fortran 95.

Examples

The following example declares an array NAMES containing 100 32-character elements, an array SOCSEC containing 100 9-character elements, and a variable NAMETY that is 10 characters long and has an initial value of `'ABCDEFGHIJ'`.

```
CHARACTER*32 NAMES(100),SOCSEC(100)*9,NAMETY*10 /'ABCDEFGHIJ'/
```

The following example includes a CHARACTER statement declaring two 8-character variables, LAST and FIRST.

```
INTEGER, PARAMETER :: LENGTH=4
CHARACTER*(4+LENGTH) LAST, FIRST
```

The following example shows a CHARACTER statement declaring an array LETTER containing 26 one-character elements. It also declares a dummy argument BUBBLE that has a passed length defined by the calling program.

```
SUBROUTINE S1(BUBBLE)
CHARACTER LETTER(26), BUBBLE*(*)
```

In the following example, NAME2 is an automatic object:

```
SUBROUTINE AUTO_NAME(NAME1)
   CHARACTER(LEN = *)           NAME1
   CHARACTER(LEN = LEN(NAME1)) NAME2
```

For More Information:

- On asterisk length specifications, see Sections Section 3.5.1.1 and Section 8.8.4.
- On the general form and rules for type declaration statements, see Section 5.1.
- On obsolescent features in Fortran 95, see Appendix A.

5.1.3. Declaration Statements for Derived Types

The derived-type (TYPE) declaration statement specifies the properties of objects and functions of derived (user-defined) type.

The derived type must be defined before you can specify objects of that type in a TYPE type declaration statement.

An object of derived type must not have the PUBLIC attribute if its type is PRIVATE.

A structure constructor specifies values for derived-type objects.

Examples

The following are examples of derived-type declaration statements:

```
TYPE(EMPLOYEE) CONTRACT
...
TYPE(SETS), DIMENSION(:,:), ALLOCATABLE :: SUBSET_1
```

The following example shows a public type with private components:

```
TYPE LIST_ITEMS
 PRIVATE
 ...
 TYPE(LIST_ITEMS), POINTER :: NEXT, PREVIOUS
END TYPE LIST_ITEMS
```

For More Information:

- On derived data types, see Section 3.3.
- On the general form and rules for type declaration statements, see Section 5.1.
- On use and host association, see Section 15.5.1.2.
- On the PUBLIC and PRIVATE attributes, see Section 5.16.
- On structure constructors, see Section 3.3.4.

5.1.4. Declaration Statements for Arrays

An array declaration (or array declarator) declares the shape of an array. It takes the following form:

```
(a-spec)
```

a-spec

Is one of the following array specifications:

- Explicit-shape (see Section 5.1.4.1)
- Assumed-shape (see Section 5.1.4.2)
- Assumed-size (see Section 5.1.4.3)
- Deferred-shape (see Section 5.1.4.4)

The array specification can be appended to the name of the array when the array is declared.

Examples

The following examples show array declarations:

```
SUBROUTINE SUB(N, C, D, Z)
  REAL, DIMENSION(N, 15) :: IARRY       ! An explicit-shape array
  REAL C(:), D(0:)                      ! An assumed-shape array
  REAL, POINTER :: B(:,:)               ! A deferred-shape array pointer
  REAL, ALLOCATABLE, DIMENSION(:) :: K  ! A deferred-shape allocatable
 array
  REAL :: Z(N,*)                        ! An assumed-size array
```

For More Information:

On the general form and rules for type declaration statements, see Section 5.1.

5.1.4.1. Explicit-Shape Specifications

An **explicit-shape array** is declared with explicit values for the bounds in each dimension of the array. An explicit-shape specification takes the following form:

```
([dl:] du[, [dl:] du]...)
```

dl

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

du

Is a specification expression indicating the upper bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

The bounds can be specified as constant or nonconstant expressions, as follows:

- If the bounds are constant expressions, the subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds. If the lower bound is greater than the upper bound, the range is empty, the extent in that dimension is zero, and the array has a size of zero.
- If the bounds are nonconstant expressions, the array must be declared in a procedure. The bounds can have different values each time the procedure is executed, since they are determined when the procedure is entered.

  The bounds are not affected by any redefinition or undefinition of the variables in the specification expression that occurs while the procedure is executing.

  The following explicit-shape arrays can specify nonconstant bounds:

  - An automatic array (the array is a local variable)
  - An adjustable array (the array is a dummy argument to a subprogram)

The following are examples of explicit-shape specifications:

```
INTEGER I(3:8, -2:5)          ! Rank-two array; range of dimension one is
...                           ! 3 to 8, range of dimension two is -2 to 5
SUBROUTINE SUB(A, B, C)
  INTEGER :: B, C
  REAL, DIMENSION(B:C) :: A  ! Rank-one array; range is B to C
```

Automatic Arrays

An **automatic array** is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The following example shows automatic arrays:

```
SUBROUTINE SUB1 (A, B)
  INTEGER A, B, LOWER
  COMMON /BOUND/ LOWER
  ...
  INTEGER AUTO_ARRAY1(B)
  ...
  INTEGER AUTO_ARRAY2(LOWER:B)
  ...
  INTEGER AUTO_ARRAY3(20, B*A/2)
END SUBROUTINE
```

Adjustable Arrays

An **adjustable array** is an explicit-shape array that is a dummy argument to a subprogram. At least one bound of an adjustable array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The array specification can contain integer variables that are either dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument specified in the bounds must be associated with an actual argument. If the specification includes a variable in a common block, the variable must have a defined value. The array specification is evaluated using the values of the actual arguments, as well as any constants or common block variables that appear in the specification.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

To avoid possible errors in subscript evaluation, make sure that the bounds expressions used to declare multidimensional adjustable arrays match the bounds as declared by the caller.

In the following example, the function computes the sum of the elements of a rank-two array. Notice how the dummy arguments M and N control the iteration:

```
  FUNCTION THE_SUM(A, M, N)
    DIMENSION A(M, N)
    SUMX = 0.0
    DO J = 1, N
      DO I = 1, M
        SUMX = SUMX + A(I, J)
      END DO
```

```
    END DO
    THE_SUM = SUMX
  END FUNCTION
```

The following are examples of calls on THE_SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = THE_SUM(A1,10,35)
SUM2 = THE_SUM(A2,3,56)
```

The following example shows how the array bounds determined when the procedure is entered do not change during execution:

```
DIMENSION ARRAY(9,5)
L = 9
M = 5
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
  DIMENSION X(-I/2:I/2,J)
  X(I/2,J) = 999
  J = 1
  I = 2
END
```

The assignments to I and J do not affect the declaration of adjustable array X as X(–4:4,5) on entry to subroutine SUB.

For More Information:

On specification expressions, see Section 4.1.7.2.

5.1.4.2. Assumed-Shape Specifications

An **assumed-shape array** is a dummy argument array that assumes the shape of its associated actual argument array. An assumed-shape specification takes the following form:

```
([dl]:[, [dl]:]...)
```

dl

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

The rank of the array is the number of colons (:) specified.

The value of the upper bound is the extent of the corresponding dimension of the associated actual argument array + *lower-bound* – 1.

The following is an example of an assumed-shape specification:

```
INTERFACE
  SUBROUTINE SUB(M)
```

```
    INTEGER M(:, 1:, 5:)
  END SUBROUTINE
END INTERFACE
INTEGER L(20, 5:25, 10)
CALL SUB(L)

SUBROUTINE SUB(M)
  INTEGER M(:, 1:, 5:)
END SUBROUTINE
```

Array M has the same extents as array L, but array M has bounds (1:20, 1:21, 5:14).

Note that an explicit interface is required when calling a routine that expects an assumed-shape or pointer array.

5.1.4.3. Assumed-Size Specifications

An **assumed-size array** is a dummy argument array that assumes the size (only) of its associated actual argument array; the rank and extents can differ for the actual and dummy arrays. An assumed-size specification takes the following form:

```
([expli-shape-spec,] [expli-shape-spec,]... [dl:] *)
```

`expli-shape-spec`

Is an explicit-shape specification (see Section 5.1.4.1).

`dl`

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

`*`

Is the upper bound of the last dimension.

The rank of the array is the number of explicit-shape specifications plus 1.

The size of the array is assumed from the actual argument associated with the assumed-size dummy array as follows:

- If the actual argument is an array of type other than default character, the size of the dummy array is the size of the actual array.
- If the actual argument is an array element of type other than default character, the size of the dummy array is a + 1 − s, where *s* is the subscript order value and *a* is the size of the actual array.
- If the actual argument is a default character array, array element, or array element substring, and it begins at character storage unit *b* of an array with *n* character storage units, the size of the dummy array is as follows:

  MAX(INT((n + 1 − b) ÷ y), 0)

  The *y* is the length of an element of the dummy array.

An assumed-size array can only be used as a whole array reference in the following cases:

- When it is an actual argument in a procedure reference that does not require the shape
- In the intrinsic function LBOUND

Because the actual size of an assumed-size array is unknown, an assumed-size array cannot be used as any of the following in an I/O statement:

- An array name in the I/O list
- A unit identifier for an internal file
- A run-time format specifier

The following is an example of an assumed-size specification:

```
SUBROUTINE SUB(A, N)
  REAL A, N
  DIMENSION A(1:N, *)
  ...
```

For More Information:

On array element order, see Section 3.5.2.2.

5.1.4.4. Deferred-Shape Specifications

A **deferred-shape array** is an array pointer or an allocatable array.

The array specification contains a colon (:) for each dimension of the array. No bounds are specified. The bounds (and shape) of allocatable arrays and array pointers are determined when space is allocated for the array during program execution.

An **array pointer** is an array declared with the POINTER attribute. Its bounds and shape are determined when it is associated with a target by pointer assignment, or when the pointer is allocated by execution of an ALLOCATE statement.

In pointer assignment, the lower bound of each dimension of the array pointer is the result of the LBOUND intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND intrinsic function applied to the corresponding dimension of the target.

A pointer dummy argument can be associated only with a pointer actual argument. An actual argument that is a pointer can be associated with a nonpointer dummy argument.

A function result can be declared to have the pointer attribute.

An **allocatable array** is declared with the ALLOCATABLE attribute. Its bounds and shape are determined when the array is allocated by execution of an ALLOCATE statement.

The following are examples of deferred-shape specifications

```
REAL, ALLOCATABLE :: A(:,:)       ! Allocatable array
REAL, POINTER :: C(:), D (:,:,:)  ! Array pointers
```

For More Information:

- On the POINTER attribute, see Section 5.15.
- On the ALLOCATABLE attribute, see Section 5.2.
- On the ALLOCATE statement, see Section 6.2.
- On pointer assignment, see Section 4.2.3.
- On the LBOUND intrinsic function, see Section 9.4.79.
- On the UBOUND intrinsic function, see Section 9.4.161.

5.2. ALLOCATABLE Attribute and Statement

The ALLOCATABLE attribute specifies that an array is an allocatable array with a deferred shape. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array.

The ALLOCATABLE attribute can be specified in a type declaration statement or an ALLOCATABLE statement, and takes one of the following forms:

```
type, [att-ls,] ALLOCATABLE [,att-ls] :: a[(d-spec)] [,a[(d-spec)]]...

ALLOCATABLE [::] a[(d-spec)] [,a[(d-spec)]]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`a`

Is the name of the allocatable array; it must not be a dummy argument or function result.

`d-spec`

Is a deferred-shape specification (: [,:]...). Each colon represents a dimension of the array.

Rules and Behavior

If the array is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

When the allocatable array is no longer needed, it can be deallocated by execution of a DEALLOCATE statement.

An allocatable array cannot be specified in a COMMON, EQUIVALENCE, DATA, or NAMELIST statement.

Allocatable arrays are not saved by default. If you want to retain the values of an allocatable array across procedure calls, you must specify the SAVE attribute for the array.

Examples

The following example shows a type declaration statement specifying the ALLOCATABLE attribute:

```
REAL, ALLOCATABLE :: Z(:, :, :)
```

The following is an example of the ALLOCATABLE statement:

```
REAL A, B(:)
ALLOCATABLE :: A(:,:), B
```

For More Information:

- On type declaration statements, see Section 5.1.
- On the ALLOCATE statement, see Section 6.2.
- On the DEALLOCATE statement, see Section 6.3.
- On allocation status, see Section 6.2.1.
- On compatible attributes, see Table 5.1.

5.3. AUTOMATIC and STATIC Attributes and Statements

The AUTOMATIC and STATIC attributes control the storage allocation of variables in subprograms.

The AUTOMATIC and STATIC attributes can be specified in a type declaration statement or an AUTOMATIC or STATIC statement, and take one of the following forms:

```
type, [att-ls,] AUTOMATIC [,att-ls] :: v [,v]...
type, [att-ls,] STATIC [,att-ls] :: v [,v]...

AUTOMATIC v [,v]...
STATIC v [,v]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`v`

Is the name of a variable or an array specification. It can be of any type.

Rules and Behavior

AUTOMATIC and STATIC declarations only affect how data is allocated in storage, as follows:

- A variable declared as AUTOMATIC and allocated in memory resides in the stack storage area.
- A variable declared as STATIC and allocated in memory resides in the static storage area.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

Automatic variables can reduce memory use because only the variables currently being used are allocated to memory.

Automatic variables allow possible recursion. With recursion, a subprogram can call itself (directly or indirectly), and resulting values are available upon a subsequent call or return to the subprogram. For recursion to occur, RECURSIVE must be specified as one of the following:

- A keyword in a FUNCTION or SUBROUTINE statement
- A compiler option
- An option in an OPTIONS statement

By default, the compiler allocates local variables of non-recursive subprograms, except for allocatable arrays, in the static storage area. The compiler may choose to allocate a variable in temporary (stack or register) storage if it notices that the variable is always defined before use. Appropriate use of the SAVE attribute can prevent compiler warnings if a variable is used before it is defined.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE (in one of the ways mentioned above).

To override any compiler option that may affect variables, explicitly specify the variables as AUTOMATIC or STATIC.

Note

Variables that are data-initialized, and variables in COMMON and SAVE statements are always static. This is regardless of whether a compiler option specifies recursion.

A variable cannot be specified as AUTOMATIC or STATIC more than once in the same scoping unit.

If the variable is a pointer, AUTOMATIC or STATIC apply only to the pointer itself, not to any associated target.

Some variables cannot be specified as AUTOMATIC or STATIC. The following table shows these restrictions:

Variable	**AUTOMATIC**	**STATIC**
Dummy argument	No	No
Automatic object	No	No
Common block item	No	Yes
Use-associated item	No	No
Function result	No	No
Component of a derived type	No	No

A variable can be specified with both the STATIC and SAVE attributes.

If a variable is in a module's outer scope, it can be specified as STATIC, but not as AUTOMATIC.

Examples

The following examples show type declaration statements specifying the AUTOMATIC and STATIC attributes:

```
REAL, AUTOMATIC :: A, B, C
INTEGER, STATIC :: ARRAY_A
```

The following example shows an AUTOMATIC AND STATIC statement:

```
...
CONTAINS
 INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
   STATIC C, D, E
   ...
 END FUNCTION
...
```

For More Information:

- On type declaration statements, see Section 5.1.
- On subprograms, see Section 8.5.
- On specifying recursive subprograms, see Section 8.5.1.1.
- On the OPTIONS statement, see Section 13.3.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compatible attributes, see Table 5.1.
- On the SAVE attribute, see Section 5.17.
- On pointers, see Section 5.15.
- On modules, see Section 8.3.

5.4. COMMON Statement

A COMMON statement defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items.

Common blocks can be named or unnamed (a **blank common**).

The COMMON statement takes the following form:

```
COMMON [/[cname]/] var-list [[,] /[cname]/ var-list]...
```

cname

Is the name of the common block. The name can be omitted for blank common (//).

var-list

Is a list of variable names, separated by commas.

The variable must not be a dummy argument, allocatable array, automatic object, function, function result, or entry to a procedure. It must not have the PARAMETER attribute. If an object of derived type is specified, it must be a sequence type.

Rules and Behavior

A common block is a global entity, and must not have the same name as any other global entity in the program, such as a subroutine or function.

Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name.

A variable can appear in only one common block within a scoping unit.

If an array is specified, it can be followed by an explicit-shape array specification. The array must not have the POINTER attribute and each bound in the specification must be a constant specification expression.

A pointer can only be associated with pointers of the same type and kind parameters, and rank.

An object with the TARGET attribute can only be associated with another object with the TARGET attribute and the same type and kind parameters.

A nonpointer can only be associated with another nonpointer, but association depends on their types, as follows:

Type of Variable	Type of Associated Variable
Intrinsic numeric[1] or numeric sequence[2]	Can be of any of these types
Default character or character sequence[2]	Can be of either of these types
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1]Default integer, default real, double precision real, default complex, double complex, or default logical.

[2]If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated directly in the common list.

So, variables can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20) REAL Y(20) COMMON /QUANTA/ A, Y
```

When common blocks from different program units have the same name, they share the same storage area when the units are combined into an executable program.

Entities are assigned storage in common blocks on a one-for-one basis. So, the data type of entities assigned by a COMMON statement in one program unit should agree with the data type of entities placed in a common block by another program unit. For example:

Program Unit A	Program Unit B

`COMMON CENTS`
`. . .`
	`INTEGER(2) MONEY`
	`COMMON MONEY`
	
	`. . .`

When these program units are combined into an executable program, incorrect results can occur if the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

Named common blocks must be declared to have the same size in each program unit. Blank common can have different lengths in different program units.

A variable or COMMON block must be declared VOLATILE if it can be read or written in a way that is not visible to the compiler.

Examples

In the following example, the COMMON statement in the main program puts HEAT and X in blank common, and KILO and Q in a named common block, BLK1:

Main Program	Subprogram

`COMMON HEAT,X /BLK1/KILO,Q`
`. . .`
 	
`CALL FIGURE`
 	
`. . .`
	`SUBROUTINE FIGURE`
	`COMMON /BLK1/LIMA,R / /ALFA,BET`
	`. . .`
	
	`RETURN`
	`END`

The COMMON statement in the subroutine makes ALFA and BET share the same storage location as HEAT and X in blank common. It makes LIMA and R share the same storage location as KILO and Q in BLK1.

The following example shows how a COMMON statement can be used to declare arrays:

```
COMMON / MIXED / SPOTTED(100), STRIPED(50,50)
```

For More Information:

- On specification expressions, see Section 4.1.7.2.
- On storage association, see Section 15.5.3.
- On derived types, see Section 3.3.
- On the EQUIVALENCE statement, see Section 5.7.
- On the interaction between COMMON and EQUIVALENCE statements, see Section 5.7.3.

- On alignment of data items in common blocks, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the VOLATILE attribute and statement, see Section 5.19.

5.5. DATA Statement

The DATA statement assigns initial values to variables before program execution. It takes the following form:

```
DATA var-list /c-list/[[,] var-list /c-list/]...
```

var-list

Is a list of variables or implied-do lists, separated by commas.

Subscript expressions and expressions in substring references must be initialization expressions.

An implied-do list in a DATA statement takes the following form:

```
(do-list, var = expr1, expr2 [,expr3])
```

do-list

Is a list of one or more array elements, substrings, scalar structure components, or implied-do lists, separated by commas. Any array elements or scalar structure components must not have a constant parent.

var

Is the name of a scalar integer variable (the implied-do variable).

expr

Are scalar integer expressions. The expressions can contain variables of other implied-do lists that have this implied-do list within their ranges.

c-list

Is a list of constants (or names of constants), or for pointer objects, NULL ( ); constants must be separated by commas. If the constant is a structure constructor, each component must be an initialization expression. If the constant is in binary, octal, or hexadecimal form, the corresponding object must be of type integer.

A constant can be specified in the form r*constant, where *r* is a repeat specification. It is a nonnegative scalar integer constant (with no kind parameter). If it is a named constant, it must have been declared previously in the scoping unit or made accessible through use or host association. If *r* is omitted, it is assumed to be 1.

Rules and Behavior

A variable can be initialized only once in an executable program. A variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The number of constants in *c-list* must equal the number of variables in *var-list*. The constants are assigned to the variables in the order in which they appear (from left to right).

The following objects cannot be initialized in a DATA statement:

- Dummy argument
- Function
- Function result
- Automatic object
- Allocatable array
- Variable that is accessible by use or host association
- Variable in a named common block (unless the DATA statement is in a block data program unit)
- Variable in blank common

Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement. You can confirm this property by specifying the variable in a SAVE statement or a type declaration statement containing the SAVE attribute.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array in the order of subscript progression. The associated constant list must contain enough values to fill the array.

Array element values can be initialized in three ways: by name, by element, or by an implied-do list (interpreted in the same way as a DO construct).

The following conversion rules and restrictions apply to variable and constant list items:

- If the constant and the variable are both of numeric type, the following conversion occurs:
    - The constant value is converted to the data type of the variable being initialized, if necessary.
    - When a binary, octal, or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the data item. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left.
- If the constant and the variable are both of character type, the following conversion occurs:
    - If the length of the constant is less than the length of the variable, the rightmost character positions of the variable are initialized with blank characters.
    - If the length of the constant is greater than the length of the variable, the character constant is truncated on the right.
- If the constant is of numeric type and the variable is of character type, the following restrictions apply:
    - The character variable must have a length of one character.

- The constant must be an integer, binary, octal, or hexadecimal constant, and must have a value in the range 0 through 255.

When the constant and variable conform to these restrictions, the variable is initialized with the character that has the ASCII code specified by the constant. (This lets you initialize a character object to any 8-bit ASCII code).

- If the constant is a Hollerith or character constant, and the variable is a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the data item.

  If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with blank characters. If the constant contains more characters than can be stored, the constant is truncated on the right.

Examples

The following example shows the three ways that DATA statements can initialize array element values:

```
DIMENSION A(10,10)

DATA A/100*1.0/    ! initialization by name

DATA A(1,1), A(10,1), A(3,3) /2*2.5, 2.0/ ! initialization by element

DATA ((A(I,J), I=1,5,2), J=1,5) /15*1.0/  ! initialization by implied-do
 list
```

The following example shows DATA statements containing structure components:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) MAN_NAME, CON_NAME
DATA MAN_NAME / EMPLOYEE(417, 'Henry Adams') /
DATA CON_NAME%ID, CON_NAME%NAME /891, "David James"/
```

In the following example, the first DATA statement assigns zero to all 10 elements of array A, and four asterisks followed by two blanks to the character variable STARS:

```
INTEGER A(10), B(10)
CHARACTER BELL, TAB, LF, FF, STARS*6
DATA A,STARS /10*0,'****'/
DATA BELL,TAB,LF,FF /7,9,10,12/
DATA (B(I), I=1,10,2) /5*1/
```

In this case, the second DATA statement assigns ASCII control character codes to the character variables BELL, TAB, LF, and FF. The last DATA statement uses an implied-do list to assign the value 1 to the odd-numbered elements in the array B.

As a Fortran 95 feature, a pointer can be initialized as disassociated by using a DATA statement. For example:

```
INTEGER, POINTER :: P
DATA P/NULL()/
```

```
END
```

For More Information:

- On implied-do lists, see Section 10.2.2.
- On initialization and specification expressions, see Section 4.1.7.
- On type declaration statements, see Section 5.1.

5.6. DIMENSION Attribute and Statement

The DIMENSION attribute specifies that an object is an array, and defines the shape of the array.

The DIMENSION attribute can be specified in a type declaration statement or a DIMENSION statement, and takes one of the following forms:

```
type, [att-ls,] DIMENSION (a-spec) [,att-ls] :: a[(a-spec)] [,a[(a-
spec)]]...

DIMENSION [::] a(a-spec) [,a(a-spec)]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`a-spec`

Is an array specification.

In a type declaration statement, any array specification following an array overrides any array specification following DIMENSION.

`a`

Is the name of the array being declared.

Rules and Behavior

The DIMENSION attribute allocates a number of storage elements to each array named, one storage element to each array element in each dimension. The size of each storage element is determined by the data type of the array.

The total number of storage elements assigned to an array is equal to the number produced by multiplying together the number of elements in each dimension in the array specification. For example, the following statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

An array can also be declared in the following statements: ALLOCATABLE, POINTER, TARGET, and COMMON.

Examples

The following examples show type declaration statements specifying the DIMENSION attribute:

```
REAL, DIMENSION(10, 10) :: A, B, C(10, 15)  ! Specification following C
                                            ! overrides the one following
                                            ! DIMENSION
REAL, ALLOCATABLE, DIMENSION(:) :: E
```

The following are examples of the DIMENSION statement:

```
DIMENSION BOTTOM(12,24,10)
DIMENSION X(5,5,5), Y(4,85), Z(100)
DIMENSION MARK(4,4,4,4)

SUBROUTINE APROC(A1,A2,N1,N2,N3)
DIMENSION A1(N1:N2), A2(N3:*)

CHARACTER(LEN = 20) D
DIMENSION A(15), B(15, 40), C(-5:8, 7), D(15)
```

For More Information:

- On type declaration statements, see Section 5.1.
- On arrays, see Section 3.5.2.
- On array specifications, see Section 5.1.4.
- On compatible attributes, see Table 5.1.
- On the ALLOCATABLE statement, see Section 5.2.
- On the COMMON statement, see Section 5.4.
- On the POINTER statement, see Section 5.15.
- On the TARGET statement, see Section 5.18.

5.7. EQUIVALENCE Statement

The EQUIVALENCE statement specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area.

The EQUIVALENCE statement takes the following form:

```
EQUIVALENCE (equiv-list) [,(equiv-list)]...
```

`equiv-list`

Is a list of two or more variables, array elements, or substrings, separated by commas (also called an equivalence set). If an object of derived type is specified, it must be a sequence type. Objects cannot have the TARGET attribute.

Each expression in a subscript or a substring reference must be an integer initialization expression. A substring must not have a length of zero.

Rules and Behavior

The following objects cannot be specified in EQUIVALENCE statements:

- Dummy argument
- Allocatable array
- Pointer
- Object of nonsequence derived type
- Object of sequence derived type containing a pointer in the structure
- Function, entry, or result name
- Named constant
- Structure component
- Subobject of any of the above objects

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

Association of objects depends on their types, as follows:

Type of Object	Type of Associated Object
Intrinsic numeric[1] or numeric sequence	Can be of any of these types
Default character or character sequence	Can be of either of these types[2]
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1]Default integer, default real, double precision real, default complex, double complex, or default logical.
[2]The lengths do not have to be equal.

So, objects can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
EQUIVALENCE(A, Y)
```

Objects of default character do not need to have the same length. The following example associates character variable D with the last 4 (of the 6) characters of character array F:

```
CHARACTER(LEN=4) D
CHARACTER(LEN=3) F(2)
EQUIVALENCE(D, F(1)(3:))
```

Entities having different data types can be associated because multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

The same storage unit cannot occur more than once in a storage sequence, and consecutive storage units cannot be specified in a way that would make them nonconsecutive.

Examples

The following EQUIVALENCE statement is invalid because it specifies the same storage unit for X(1) and X(2):

```
REAL, DIMENSION(2), :: X
REAL :: Y
EQUIVALENCE(X(1), Y), (X(2), Y)
```

The following EQUIVALENCE statement is invalid because because A(1) and A(2) will not be consecutive:

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

In the following example, the EQUIVALENCE statement causes the four elements of the integer array IARR to share the same storage as that of the double-precision variable DVAR.

```
DOUBLE PRECISION DVAR
INTEGER(KIND=2) IARR(4)
EQUIVALENCE(DVAR, IARR(1))
```

In the following example, the EQUIVALENCE statement causes the first character of the character variables KEY and STAR to share the same storage location. The character variable STAR is equivalent to the substring KEY(1:10).

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE(KEY, STAR)
```

For More Information:

- On initialization expressions, see Section 4.1.7.1.
- On derived data types, see Section 3.3.
- On storage units, sequence, and association, see Section 15.5.3.

5.7.1. Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

Two or more elements of the same array should not be associated with each other in one or more EQUIVALENCE statements. For example, you cannot use an EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following valid example:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE(TABLE(2,2), TRIPLE(1,2,2))
```

These statements cause the entire array TABLE to share part of the storage allocated to TRIPLE. Table 5.3 shows how these statements align the arrays.

Table 5.3. Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Element	**Element Number**	**Array Element**	**Element Number**
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Each of the following statements also aligns the two arrays as shown in Table 5.3:

```
EQUIVALENCE(TABLE, TRIPLE(2,2,1))
EQUIVALENCE(TRIPLE(1,1,2), TABLE(2,1))
```

You can also make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the following statement:

```
EQUIVALENCE(A(3,4), B(2,4))
```

The entire array A shares part of the storage allocated to array B. Table 5.4 shows how these statements align the arrays. The arrays can also be aligned by the following statements:

```
EQUIVALENCE(A, B(4,1))
EQUIVALENCE(B(3,2), A(2,2))
```

Table 5.4. Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	**Element Number**	**Array Element**	**Element Number**
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (the linear element number), even though the array was defined as multidimensional. For example, the following statements align the two arrays as shown in Table 5.4:

```
DIMENSION B(2:4,1:4), A(2:3,1:4)
EQUIVALENCE(B(6), A(4))
```

5.7.2. Making Substrings Equivalent

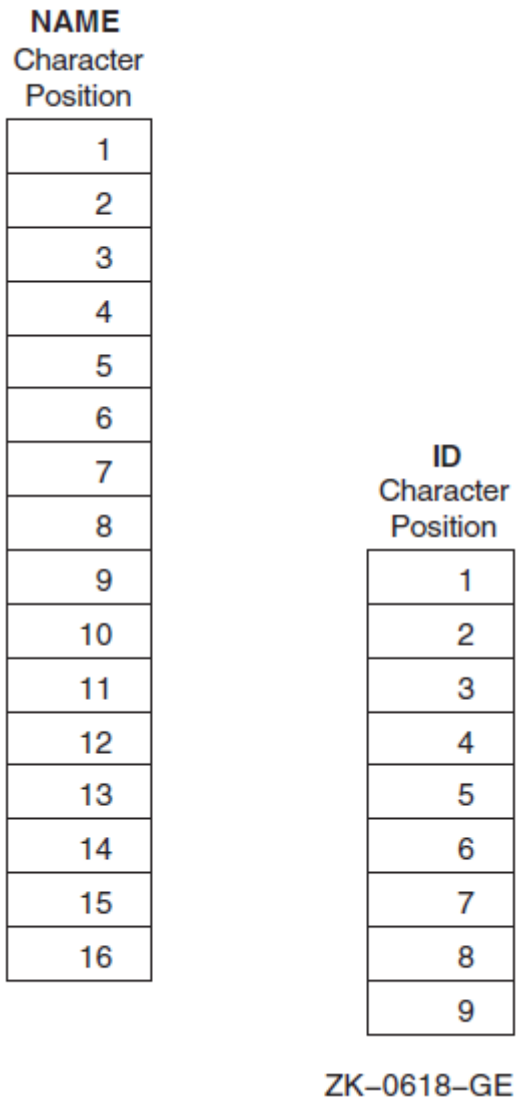
When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets associations between the other corresponding characters in the character entities; for example:

```
CHARACTER NAME*16, ID*9
EQUIVALENCE(NAME(10:13), ID(2:5))
```

These statements cause character variables NAME and ID to share space (see Figure 5.1). The arrays can also be aligned by the following statement:

```
EQUIVALENCE(NAME(9:9), ID(1:1))
```

Figure 5.1. Equivalence of Substrings

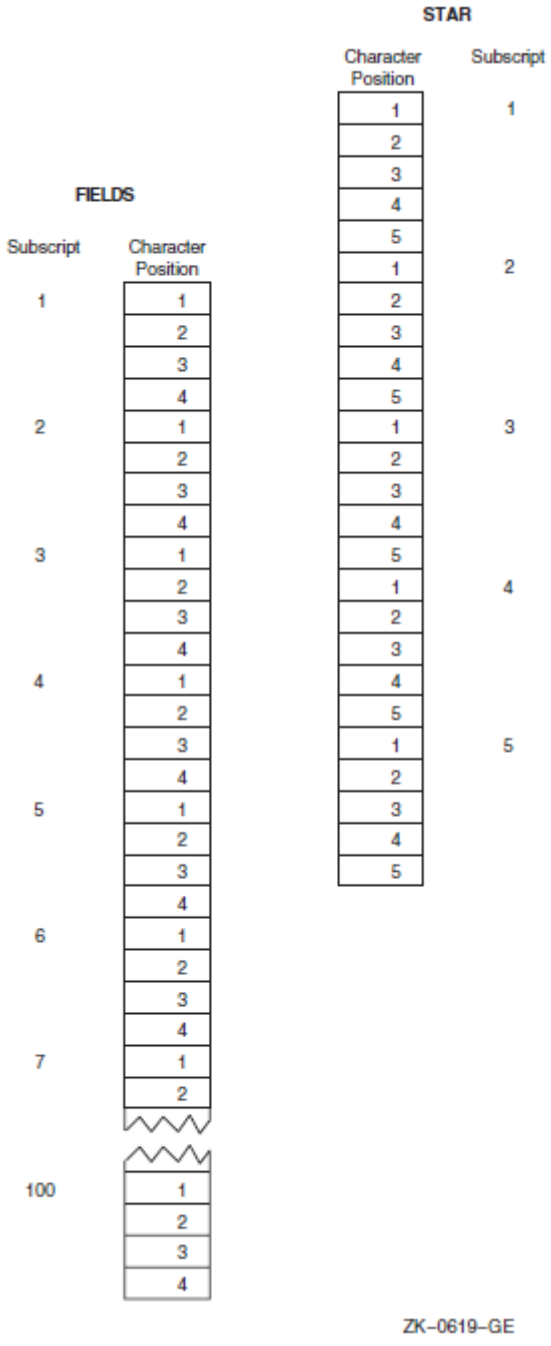


If the character substring references are array elements, the EQUIVALENCE statement sets associations between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position. For example, the following statements cause character arrays FIELDS and STAR to share storage (see Figure 5.2).

```
CHARACTER FIELDS(100)*4, STAR(5)*5
EQUIVALENCE(FIELDS(1)(2:4), STAR(2)(3:5))
```

Figure 5.2. Equivalence of Character Arrays



The EQUIVALENCE statement cannot assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array. The EQUIVALENCE statement also cannot assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

5.7.3. EQUIVALENCE and COMMON Interaction

A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

Examples

Figure 5.3 and Figure 5.4 demonstrate valid and invalid extensions of the common block, respectively.

Figure 5.3. A Valid Extension of a Common Block

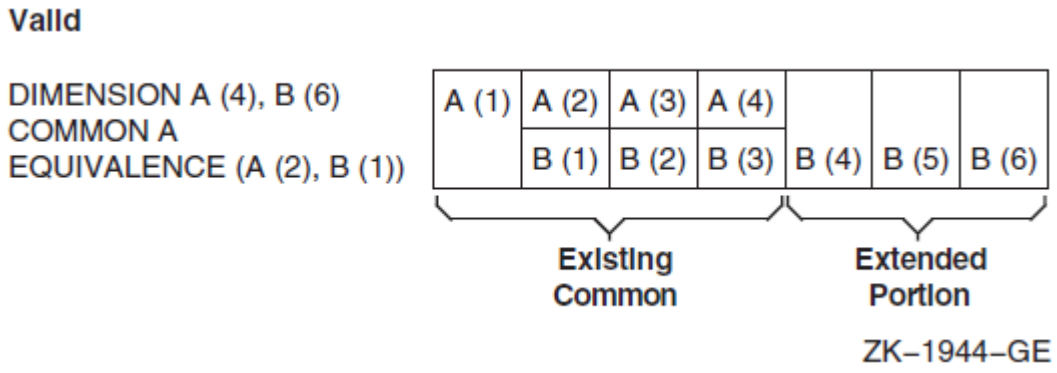
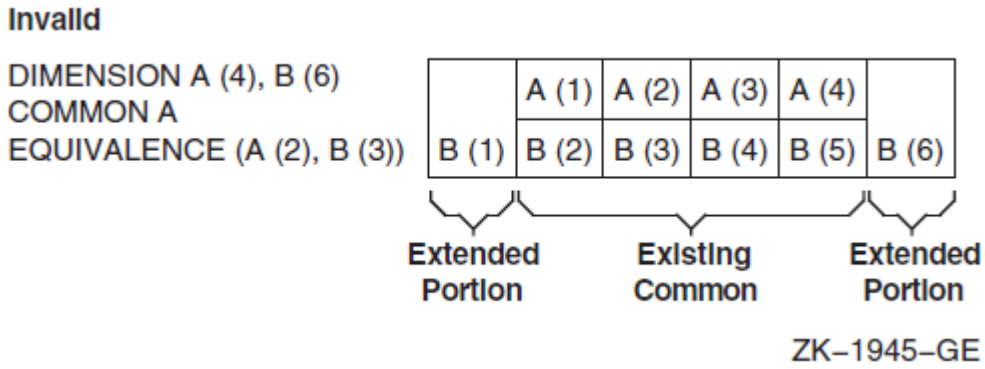


Figure 5.4. An Invalid Extension of a Common Block



The second example is invalid because the extended portion, B(1), precedes the first element of the common block.

The following example shows a valid EQUIVALENCE statement and an invalid EQUIVALENCE statement in the context of a common block.

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE(B, D(1))       ! Valid, because common block is extended
                           ! from the end.

COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE(B, D(3))       ! Invalid, because D(1) would extend common
                           ! block to precede A's location.
```

5.8. EXTERNAL Attribute and Statement

The EXTERNAL attribute allows an external or dummy procedure to be used as an actual argument. (To specify intrinsic procedures as actual arguments, use the INTRINSIC attribute).

The EXTERNAL attribute can be specified in a type declaration statement or an EXTERNAL statement, and takes one of the following forms:

```
type, [att-ls,] EXTERNAL [,att-ls] :: ex-pro [,ex-pro]...

EXTERNAL ex-pro [,ex-pro]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`ex-pro`

Is the name of an external (user-supplied) procedure or dummy procedure.

Rules and Behavior

In a type declaration statement, only *functions* can be declared EXTERNAL. However, you can use the EXTERNAL *statement* to declare subroutines and block data program units, as well as functions, to be external.

The name declared EXTERNAL is assumed to be the name of an external procedure, even if the name is the same as that of an intrinsic procedure. For example, if SIN is declared with the EXTERNAL attribute, all subsequent references to SIN are to a user-supplied function named SIN, not to the intrinsic function of the same name.

You can include the name of a block data program unit in the EXTERNAL statement to force a search of the object module libraries for the block data program unit at link time. However, the name of the block data program unit must not be used in a type declaration statement.

Examples

The following example shows type declaration statements specifying the EXTERNAL attribute:

```
PROGRAM TEST
...
INTEGER, EXTERNAL :: BETA
LOGICAL, EXTERNAL :: COS
...
CALL SUB(BETA)       ! External function BETA is an actual argument
```

You can use a name specified in an EXTERNAL statement as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a CALL statement; for example:

```
EXTERNAL FACET
CALL BAR(FACET)
```

```
SUBROUTINE BAR(F)
EXTERNAL F
CALL F(2)
```

Used as an argument, a complete function reference represents a value, not a subprogram; for example, FUNC(B) represents a value in the following statement:

```
CALL SUBR(A, FUNC(B), C)
```

For More Information:

- On type declaration statements, see Section 5.1.
- On intrinsic procedures, see Chapter 9.
- On the INTRINSIC attribute, see Section 5.11.
- On compatible attributes, see Table 5.1.
- On a compiler option that changes the interpretation of the EXTERNAL statement, see Section B.4.

5.9. IMPLICIT Statement

The IMPLICIT statement overrides the default implicit typing rules for names. (The default data type is INTEGER for names beginning with the letters I through N, and REAL for names beginning with any other letter).

The IMPLICIT statement takes one of the following forms:

```
IMPLICIT type (a[,a]...)[, type (a[,a]...)]...
IMPLICIT NONE
```

`type`

Is a data type specifier (CHARACTER*(*) is not allowed).

`a`

Is a single letter, a dollar sign ($), or a range of letters in alphabetical order. The form for a range of letters is a_1-a_2, where the second letter follows the first alphabetically (for example, A-C).

The dollar sign can be used at the end of a range of letters, since IMPLICIT interprets the dollar sign to alphabetically follow the letter Z. For example, a range of X-$ would apply to identifiers beginning with the letters X, Y, Z, or $.

Rules and Behavior

The IMPLICIT statement assigns the specified data type (and kind parameter) to all names that have no explicit data type and begin with the specified letter or range of letters. It has no effect on the default types of intrinsic procedures.

When the data type is CHARACTER*len, *len* is the length for character type. The *len* is an unsigned integer constant or an integer initialization expression enclosed in parentheses. The range for *len* is 1 to 65535.

Names beginning with a dollar sign ($) are implicitly INTEGER.

The IMPLICIT NONE statement disables all implicit typing defaults. When IMPLICIT NONE is used, all names in a program unit must be explicitly declared. An IMPLICIT NONE statement must precede any PARAMETER statements, and there must be no other IMPLICIT statements in the scoping unit.

Note

To receive diagnostic messages when variables are used but not declared, you can specify a compiler option instead of using IMPLICIT NONE.

The following IMPLICIT statement represents the default typing for names when they are not explicitly typed:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

Examples

The following are examples of the IMPLICIT statement:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL(1) (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
IMPLICIT TYPE(COLORS) (E-F), INTEGER (G-H)
```

For More Information:

On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

5.10. INTENT Attribute and Statement

The INTENT attribute specifies the intended use of one or more dummy arguments.

The INTENT attribute can be specified in a type declaration statement or an INTENT statement, and takes one of the following forms:

```
type, [att-ls,] INTENT (intent-spec) [,att-ls] :: d-arg [, d-arg]...

INTENT (intent-spec) [::] d-arg [, d-arg]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`intent-spec`

Is one of the following specifiers:

- IN

Specifies that the dummy argument will be used only to provide data to the procedure. The dummy argument must not be redefined (or become undefined) during execution of the procedure.

Any associated actual argument must be an expression.

- OUT

  Specifies that the dummy argument will be used to pass data from the procedure back to the calling program. The dummy argument is undefined on entry and must be defined before it is referenced in the procedure.

  Any associated actual argument must be definable.

- INOUT

  Specifies that the dummy argument can both provide data to the procedure and return data to the calling program.

  Any associated actual argument must be definable.

`d-arg`

Is the name of a dummy argument. It cannot be a dummy procedure or dummy pointer.

Rules and Behavior

The INTENT statement can only appear in the specification part of a subprogram or interface body.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

If a function specifies a defined operator, the dummy arguments must have intent IN.

If a subroutine specifies defined assignment, the first argument must have intent OUT or INOUT, and the second argument must have intent IN.

A dummy argument with intent IN (or a subobject of such a dummy argument) must not appear as any of the following:

- A DO variable or implied-DO variable
- The variable of an assignment statement
- The *pointer-object* of a pointer assignment statement
- An *object* or STAT= variable in an ALLOCATE or DEALLOCATE statement
- An input item in a READ statement
- A variable name in a NAMELIST statement if the namelist group name appears in a NML= specifier in a READ statement
- An internal file unit in a WRITE statement
- A definable variable in an INQUIRE statement
- An IOSTAT= or SIZE= specifier in an I/O statement

- An actual argument in a reference to a procedure with an explicit interface if the associated dummy argument has intent OUT or INOUT

If an actual argument is an array section with a vector subscript, it cannot be associated with a dummy array that is defined or redefined (has intent OUT or INOUT).

Examples

The following example shows type declaration statements specifying the INTENT attribute:

```
SUBROUTINE TEST(I, J)
  INTEGER, INTENT(IN) :: I
  INTEGER, INTENT(OUT), DIMENSION(I) :: J
```

The following are examples of the INTENT statement:

```
SUBROUTINE TEST(A, B, X)
   INTENT(INOUT) :: A, B
   ...

SUBROUTINE CHANGE(FROM, TO)
   USE EMPLOYEE_MODULE
   TYPE(EMPLOYEE) FROM, TO
   INTENT(IN) FROM
   INTENT(OUT) TO
   ...
```

For More Information:

- On type declaration statements, see Section 5.1.
- On argument association, see Section 8.8.
- On compatible attributes, see Table 5.1.

5.11. INTRINSIC Attribute and Statement

The INTRINSIC attribute allows the specific name of an intrinsic procedure to be used as an actual argument. (Not all specific names can be used as actual arguments. For more information, see Table 9.1).

The INTRINSIC attribute can be specified in a type declaration statement or an INTRINSIC statement, and takes one of the following forms:

```
type, [att-ls,] INTRINSIC [,att-ls] :: in-pro [,in-pro]...

INTRINSIC in-pro [,in-pro]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`in-pro`

Is the name of an intrinsic procedure.

Rules and Behavior

In a type declaration statement, only *functions* can be declared INTRINSIC. However, you can use the INTRINSIC *statement* to declare subroutines, as well as functions, to be intrinsic.

The name declared INTRINSIC is assumed to be the name of an intrinsic procedure. If a generic intrinsic function name is given the INTRINSIC attribute, the name retains its generic properties.

Examples

The following example shows a type declaration statement specifying the INTRINSIC attribute:

```
PROGRAM EXAMPLE
...
REAL(8), INTRINSIC :: DACOS
...
CALL TEST(X, DACOS)     ! Intrinsic function DACOS is an actual argument
```

The following example shows an INTRINSIC statement:

Main Program	**Subprogram**

`EXTERNAL CTN`
`INTRINSIC SIN, COS`
`. . .`
 	
`CALL TRIG(ANGLE,SIN,SINE)`
`. . .`
 	
`CALL TRIG(ANGLE,COS,COSINE)`
`. . .`
 	
`CALL TRIG(ANGLE,CTN,COTANGENT)`
	`SUBROUTINE TRIG(X,F,Y)`
	`Y = F(X)`
	`RETURN`
	`END`
	
	`FUNCTION CTN(X)`
	`CTN = COS(X)/SIN(X)`
	
	`RETURN`
	`END`

Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Fortran 95/90 library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

For More Information:

- On type declaration statements, see Section 5.1.
- On specific intrinsic procedures, see Chapter 9.
- On referencing generic intrinsic functions, see Section 8.8.8.1.
- On referencing elemental intrinsic procedures, see Section 8.8.8.2.
- On compatible attributes, see Table 5.1.

5.12. NAMELIST Statement

The NAMELIST statement associates a name with a list of variables. This group name can be referenced in some input/output operations.

A NAMELIST statement takes the following form:

```
NAMELIST /group/var-list [[,] /group/var-list]...
```

group

Is the name of the group.

var-list

Is a list of variables (separated by commas) that are to be associated with the preceding group name. The variables can be of any data type.

Rules and Behavior

The namelist group name is used by namelist I/O statements instead of an I/O list. The unique group name identifies a list whose entities can be modified or transferred.

A variable can appear in more than one namelist group.

Each variable in *var-list* must be accessed by use or host association, or it must have its type, type parameters, and shape explicitly or implicitly specified in the same scoping unit. If the variable is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The following variables cannot be specified in a namelist group:

- Array dummy argument with nonconstant bounds
- Variable with assumed character length
- Allocatable array
- Automatic object
- Pointer
- Variable of a type that has a pointer as an ultimate component
- Subobject of any of the above objects

Only the variables specified in the namelist can be read or written in namelist I/O. It is not necessary for the input records in a namelist input statement to define every variable in the associated namelist.

The order of variables in the namelist controls the order in which the values appear on namelist output. Input of namelist values can be in any order.

If the group name has the PUBLIC attribute, no item in the variable list can have the PRIVATE attribute.

The group name can be specified in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the group name is treated as a continuation of the list for that group name.

Examples

In the following example, D and E are added to the variables A, B, and C for group name LIST:

```
NAMELIST /LIST/ A, B, C

NAMELIST /LIST/ D, E
```

In the following example, two group names are defined:

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

Group name INPUT contains variables NAME, GRADE, and DATE. Group name OUTPUT contains variables TOTAL and NAME.

For More Information:

On namelist input, see Section 10.3.1.3; output, see Section 10.5.1.3.

5.13. OPTIONAL Attribute and Statement

The OPTIONAL attribute permits dummy arguments to be omitted in a procedure reference.

The OPTIONAL attribute can be specified in a type declaration statement or an OPTIONAL statement, and takes one of the following forms:

```
type, [att-ls,] OPTIONAL [,att-ls] :: d-arg [,d-arg]...

OPTIONAL [::] d-arg [,d-arg]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`d-arg`

Is the name of a dummy argument.

Rules and Behavior

The OPTIONAL attribute can only appear in the scoping unit of a subprogram or an interface body, and can only be specified for dummy arguments.

A dummy argument is “present” if it associated with an actual argument. A dummy argument that is not optional must be present. You can use the PRESENT intrinsic function to determine whether an optional dummy argument is associated with an actual argument.

To call a procedure that has an optional argument, you must use an explicit interface.

Examples

The following example shows a type declaration statement specifying the OPTIONAL attribute:

```
SUBROUTINE TEST(A)
```

```
  REAL, OPTIONAL, DIMENSION(-10:2) :: A
END SUBROUTINE
```

The following is an example of the OPTIONAL statement:

```
SUBROUTINE TEST(A, B, L, X)
  OPTIONAL :: B
  INTEGER A, B, L, X

  IF (PRESENT(B)) THEN        ! Printing of B is conditional
    PRINT *, A, B, L, X       !   on its presence
  ELSE
    PRINT *, A, L, X
  ENDIF
END SUBROUTINE

INTERFACE
  SUBROUTINE TEST(ONE, TWO, THREE, FOUR)
    INTEGER ONE, TWO, THREE, FOUR
    OPTIONAL :: TWO
  END SUBROUTINE
END INTERFACE

INTEGER I, J, K, L

I = 1
J = 2
K = 3
L = 4

CALL TEST(I, J, K, L)            ! Prints:  1  2  3  4
CALL TEST(I, THREE=K, FOUR=L)    ! Prints:  1  3  4
END
```

Note that in the second call to subroutine TEST, the second positional (optional) argument is omitted. In this case, all following arguments must be keyword arguments.

For More Information:

- On type declaration statements, see Section 5.1.
- On the PRESENT intrinsic function, see Section 9.4.117.
- On optional arguments, see Section 8.8.1.
- On compatible attributes, see Table 5.1.

5.14. PARAMETER Attribute and Statement

The PARAMETER attribute defines a named constant.

The PARAMETER attribute can be specified in a type declaration statement or a PARAMETER statement, and takes one of the following forms:

```
type, [att-ls,] PARAMETER [,att-ls] :: c = expr [, c = expr]...

PARAMETER [(] c = expr [, c = expr]...[)]
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`c`

Is the name of the constant.

`expr`

Is an initialization expression. It can be of any data type.

Rules and Behavior

The type, type parameters, and shape of the named constant are determined in one of the following ways:

- By an explicit type declaration statement in the same scoping unit.
- By the implicit typing rules in effect for the scoping unit. If the named constant is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

For example, consider the following statement:

```
PARAMETER (MU=1.23)
```

According to implicit typing, MU is of integer type, so MU=1. For MU to equal 1.23, it should previously be declared REAL in a type declaration or be declared in an IMPLICIT statement.

A named constant must not appear in a format specification or as the character count for Hollerith constants. For compilation purposes, writing the name is the same as writing the value.

If the named constant is used as the length specifier in a CHARACTER declaration, it must be enclosed in parentheses.

The name of a constant cannot appear as part of another constant, although it can appear as either the real or imaginary part of a complex constant.

You can only use the named constant within the scoping unit containing the defining PARAMETER statement.

Any named constant that appears in the initialization expression must have been defined previously in the same type declaration statement (or in a previous type declaration statement or PARAMETER statement), or made accessible by use or host association.

Examples

The following example shows a type declaration statement specifying the PARAMETER attribute:

```
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
```

The following is an example of the PARAMETER statement:

```
REAL(4) PI, PIOV2
REAL(8) DPI, DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME

PARAMETER (PI=3.1415927, DPI=3.141592653589793238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

For More Information:

- On type declaration statements, see Section 5.1.
- On initialization expressions, see Section 4.1.7.1.
- On the IMPLICIT statement, see Section 5.9.
- On compatible attributes, see Table 5.1.
- On an alternative syntax for the PARAMETER statement, see Section B.5.

5.15. POINTER Attribute and Statement

The POINTER attribute specifies that an object is a pointer (a dynamic variable). A pointer does not contain data, but *points* to a scalar or array variable where data is stored. A pointer has no initial storage set aside for it; memory storage is created for the pointer as a program runs.

The POINTER attribute can be specified in a type declaration statement or a POINTER statement, and takes one of the following forms:

```
type, [att-ls,] POINTER [,att-ls] :: ptr [(d-spec)] [,ptr [(d-spec)]]...

POINTER [::] ptr [(d-spec)] [,ptr [(d-spec)]]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`ptr`

Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.

`d-spec`

Is a deferred-shape specification (: [,:]...). Each colon represents a dimension of the array.

Rules and Behavior

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target. A pointer must not be referenced or defined until memory is associated with it.

Each pointer has an association status, which tells whether the pointer is currently associated with a target object. When a pointer is initially declared, its status is undefined. You can use the ASSOCIATED intrinsic function to find the association status of a pointer.

If the pointer is an array, and it is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

A pointer cannot be specified in a DATA, EQUIVALENCE, or NAMELIST statement.

Examples

The following example shows type declaration statements specifying the POINTER attribute:

```
TYPE(SYSTEM), POINTER :: CURRENT, LAST
REAL, DIMENSION(:,:), POINTER :: I, J, REVERSE
```

The following is an example of the POINTER statement:

```
TYPE(SYSTEM) :: TODAYS
POINTER :: TODAYS, A(:,:)
```

For More Information:

- On type declaration statements, see Section 5.1.
- On deferred-shape arrays, see Section 5.1.4.4.
- On compatible attributes, see Table 5.1.
- On pointer assignment, see Section 4.2.3.
- On the ALLOCATE statement, see Section 6.2.
- On pointer association, see Section 15.5.2.
- On pointer arguments, see Section 8.8.3.
- On the ASSOCIATED intrinsic function, see Section 9.4.15.
- On a different kind of POINTER statement, see Section B.11.
- On the NULL intrinsic function, which can be used to disassociate a pointer, see Section 9.4.110.

5.16. PRIVATE and PUBLIC Attributes and Statements

The PRIVATE and PUBLIC attributes specify the accessibility of entities in a module. (These attributes are also called accessibility attributes).

The PRIVATE and PUBLIC attributes can be specified in a type declaration statement or a PRIVATE or PUBLIC statement, and take one of the following forms:

```
type, [att-ls,] PRIVATE [,att-ls] :: entity [,entity]...
```

```
type, [att-ls,] PUBLIC [,att-ls] :: entity [,entity]...

PRIVATE [[::] entity [,entity]...]
PUBLIC [[::] entity [,entity]...]
```

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

entity

Is one of the following:

- Variable name
- Procedure name
- Derived type name
- Named constant
- Namelist group name

In statement form, an entity can also be a generic identifier (a generic name, defined operator, or defined assignment).

Rules and Behavior

The PRIVATE and PUBLIC attributes can only appear in the scoping unit of a module.

Only one PRIVATE or PUBLIC statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module.

If no PUBLIC or PRIVATE statements are specified in a module, the default is PUBLIC accessibility. Entities with PUBLIC accessibility can be accessed from outside the module by means of a USE statement.

If a derived type is declared PRIVATE in a module, its components are also PRIVATE. The derived type and its components are accessible to any subprograms within the defining module through host association, but the type is not accessible from outside the module and in most cases the components are not accessible, either.

However, if a public entity is declared to be of a type declared PRIVATE, the components of that type are accessible as components of the entity. For example:

```
module m2
type hidden
   integer f1,f2
end type hidden
end

module m3
   use m2
```

```
   private
   type(hidden),public :: x
  end

subroutine import
   use m3
   x%f1 = 1
   end subroutine
```

In this example, the F1 component of X is accessible, even though the type HIDDEN is PRIVATE.

If the derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

Examples

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C
    DIMENSION LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following derived-type declaration statement indicates that the type is restricted to the module:

```
TYPE, PRIVATE  :: DATA
  ...
END TYPE DATA
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

For More Information:

- On type declaration statements, see Section 5.1.
- On derived types, see Section 3.3.
- On compatible attributes, see Table 5.1.
- On generic identifiers, see Section 8.9.3.
- On modules, see Section 8.3.
- On the USE statement, see Section 8.3.2.
- On use and host association, see Section 15.5.1.2.

5.17. SAVE Attribute and Statement

The SAVE attribute causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.

The SAVE attribute can be specified in a type declaration statement or a SAVE statement, and takes one of the following forms:

```
type, [att-ls,] SAVE [,att-ls] :: [object [,object]...]

SAVE [object [,object]...]
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`object`

Is the name of an object, or the name of a common block enclosed in slashes (/ *common-block-name*/).

Rules and Behavior

In VSI Fortran, certain variables are given the SAVE attribute, or not, by default:

- The following variables are saved by default:
  - COMMON variables
  - Local variables of recursive subprograms
  - Data initialized by DATA statements

- The following variables are not saved by default:
  - Variables that are declared AUTOMATIC
  - Local variables that are allocatable arrays
  - Derived-type variables that are data initialized by default initialization of any of their components
  - RECORD variables that are data initialized by default initialization specified in its STRUCTURE declaration
- Local variables that are not described in the preceding two lists are saved by default.

To enhance portability and avoid possible compiler warning messages, VSI recommends that you use the SAVE statement to name variables whose values you want to preserve between subprogram invocations.

When a SAVE statement does not explicitly contain a list, all allowable items in the scoping unit are saved.

A SAVE statement cannot specify the following (their values cannot be saved):

- Blank common
- Object in a common block
- Procedure
- Dummy argument
- Function result
- Automatic object
- PARAMETER (named) constant

Even though a common block can be included in a SAVE statement, individual variables within the common block can become undefined (or redefined) in another scoping unit.

If a common block is saved in any scoping unit of a program (other than the main program), it must be saved in every scoping unit in which the common block appears.

A SAVE statement has no effect in a main program.

Examples

The following example shows a type declaration statement specifying the SAVE attribute:

```
SUBROUTINE TEST()
  REAL, SAVE :: X, Y
```

The following is an example of the SAVE statement:

```
SAVE A, /BLOCK_B/, C, /BLOCK_D/, E
```

For More Information:

- On type declaration statements, see Section 5.1.

- On common blocks, see Section 5.4.
- On the DATA statement, see Section 5.5.
- On recursive program units, see Section 8.5.1.1.
- On compatible attributes, see Table 5.1.
- On modules, see Section 8.3.

5.18. TARGET Attribute and Statement

The TARGET attribute specifies that an object can become the target of a pointer (it can be pointed to).

The TARGET attribute can be specified in a type declaration statement or a TARGET statement, and takes one of the following forms:

```
type, [att-ls,] TARGET [,att-ls] :: object [(a-spec)] [,object [(a-
spec)]]...

TARGET [::] object [(a-spec)] [,object [(a-spec)]]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`object`

Is the name of the object. The object must not be declared with the PARAMETER attribute.

`a-spec`

Is an array specification.

Rules and Behavior

A pointer is associated with a target by pointer assignment or by an ALLOCATE statement.

If an object does not have the TARGET attribute or has not been allocated (using an ALLOCATE statement), no part of it can be accessed by a pointer.

Examples

The following example shows type declaration statements specifying the TARGET attribute:

```
TYPE(SYSTEM), TARGET :: FIRST
REAL, DIMENSION(20, 20), TARGET :: C, D
```

The following is an example of a TARGET statement:

```
TARGET :: C(50, 50), D
```

For More Information:

- On type declaration statements, see Section 5.1.
- On the ALLOCATE statement, see Section 6.2.
- On compatible attributes, see Table 5.1.
- On pointer assignment, see Section 4.2.3.
- On pointer association, see Section 15.5.2.

5.19. VOLATILE Attribute and Statement

The VOLATILE attribute specifies that the value of an object is entirely unpredictable, based on information local to the current program unit. It prevents objects from being optimized during compilation.

The VOLATILE attribute can be specified in a type declaration statement or a VOLATILE statement, and takes one of the following forms:

```
type, [att-ls,] VOLATILE [,att-ls] :: object [,object]...

VOLATILE object [,object]...
```

`type`

Is a data type specifier.

`att-ls`

Is an optional list of attribute specifiers.

`object`

Is the name of an object, or the name of a common block enclosed in slashes.

Rules and Behavior

A variable or COMMON block must be declared VOLATILE if it can be read or written in a way that is not visible to the compiler. For example:

- If an operating system feature is used to place a variable in shared memory (so that it can be accessed by other programs), the variable must be declared VOLATILE.
- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.

If an array is declared VOLATILE, each element in the array becomes volatile. If a common block is declared VOLATILE, each variable in the common block becomes volatile.

If an object of derived type is declared VOLATILE, its components become volatile.

If a pointer is declared VOLATILE, the pointer itself becomes volatile.

A VOLATILE statement cannot specify the following:

- Procedure
- Function result
- Namelist group

Example

The following example shows a type declaration statement specifying the VOLATILE attribute:

```
INTEGER, VOLATILE :: D, E
```

The following example shows a VOLATILE statement:

```
PROGRAM TEST
LOGICAL(1) IPI(4)
INTEGER(4) A, B, C, D, E, ILOOK
INTEGER(4) P1, P2, P3, P4
COMMON /BLK1/A, B, C

VOLATILE /BLK1/, D, E
EQUIVALENCE(ILOOK, IPI)
EQUIVALENCE(A, P1)
EQUIVALENCE(P1, P4)
```

The named common block, BLK1, and the variables D and E are volatile. Variables P1 and P4 become volatile because of the direct equivalence of P1 and the indirect equivalence of P4.

For More Information:

- On type declaration statements, see Section 5.1.
- On compatible attributes, see Table 5.1.
- On optimizations performed by the compiler, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Chapter 6. Dynamic Allocation

6.1. Overview

Data objects can be static or dynamic. If a data object is static, a fixed amount of memory storage is created for it at compile time and is not freed until the program exits. If a data object is dynamic, memory storage for the object can be created (allocated), altered, or freed (deallocated) as a program executes.

In Fortran 95/90, pointers, allocatable arrays, and automatic arrays are dynamic data objects.

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target. A pointer can be dynamically disassociated from a target by using a NULLIFY statement.

An ALLOCATE statement can also be used to create storage for an allocatable array. A DEALLOCATE statement is used to free the storage space reserved in a previous ALLOCATE statement.

Automatic arrays differ from allocatable arrays in that they are automatically allocated and deallocated whenever you enter or leave a procedure, respectively.

For More Information:

- On pointer assignment, see Section 4.2.3.
- On automatic arrays, see Section 5.1.4.1.
- On the NULL intrinsic function, which can also be used to disassociate a pointer, see Section 9.4.110.

6.2. ALLOCATE Statement

The ALLOCATE statement dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized.

The ALLOCATE statement takes the following form:

```
ALLOCATE (object [(s-spec[,s-spec...])] [,object[(s-spec[,s-spec...])]]...
         [,STAT=sv])
```

`object`

Is the object to be allocated. It is a variable name or structure component, and must be a pointer or allocatable array. The object can be of type character with zero length.

`s-spec`

Is a shape specification in the form [lower-bound:]upper-bound. Each bound must be a scalar integer expression. The number of shape specifications must be the same as the rank of the *object*.

`sv`

Is a scalar integer variable in which the status of the allocation is stored.

Rules and Behavior

A bound in *s-spec* must not be an expression containing an array inquiry function whose argument is any allocatable object in the same ALLOCATE statement; for example, the following is not permitted:

```
INTEGER ERR
INTEGER, ALLOCATABLE :: A(:), B(:)
...
ALLOCATE(A(10:25), B(SIZE(A)), STAT=ERR)  ! A is invalid as an argument
                                          !   to function SIZE
```

If a STAT variable is specified, it must not be allocated in the ALLOCATE statement in which it appears. If the allocation is successful, the variable is set to zero. If the allocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error). If no STAT variable is specified and an error condition occurs, program execution terminates.

Examples

The following is an example of the ALLOCATE statement:

```
INTEGER J, N, ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE(A(0:80), B(-3:J+1, N), STAT = ALLOC_ERR)
```

For More Information:

- On allocatable arrays, see Section 5.2.
- On pointers, see Section 5.15.
- On run-time error messages, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/] or online documentation.

6.2.1. Allocation of Allocatable Arrays

The bounds (and shape) of an allocatable array are determined when it is allocated. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification.

If the lower bound is greater than the upper bound, that dimension has an extent of zero, and the array has a size of zero. If the lower bound is omitted, it is assumed to be 1.

When an array is allocated, it is definable. If you try to allocate a currently allocated allocatable array, an error occurs.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
REAL, ALLOCATABLE :: E(:,:)
...
IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7))
```

Allocation Status

During program execution, the allocation status of an allocatable array is one of the following:

- Not currently allocated

  The array was never allocated or the last operation on it was a deallocation. Such an array must not be referenced or defined.

- Currently allocated

  The array was allocated by an ALLOCATE statement. Such an array can be referenced, defined, or deallocated.

If an allocatable array has the SAVE attribute, it has an initial status of “not currently allocated.” If the array is then allocated, its status changes to “currently allocated.” It keeps that status until the array is deallocated.

If an allocatable array *does not* have the SAVE attribute, it has the status of “not currently allocated” at the beginning of each invocation of the procedure. If the array's status changes to “currently allocated”, it is deallocated if the procedure is terminated by execution of a RETURN or END statement.

Examples

Example 6.1 shows a program that performs virtual memory allocation. This program uses Fortran 95/90 standard-conforming statements instead of calling an operating system memory allocation routine.

Example 6.1. Allocating Virtual Memory

```
! Program accepts an integer and displays square root values

  INTEGER(4) :: N
  READ (5,*) N                          ! Reads an integer value
  CALL MAT(N)
  END

! Subroutine MAT uses the typed integer value to display the square
! root values of numbers from 1 to N (the number read)

  SUBROUTINE MAT(N)
  REAL(4), ALLOCATABLE :: SQR(:)        ! Declares SQR as a one-dimensional
                                        !          allocatable array
  ALLOCATE (SQR(N))                     ! Allocates array SQR

  DO J=1,N
     SQR(J) = SQRT(FLOATJ(J))           ! FLOATJ converts integer to REAL
  ENDDO

  WRITE (6,*) SQR                       ! Displays calculated values
  DEALLOCATE (SQR)                      ! Deallocates array SQR
  END SUBROUTINE MAT
```

For More Information:

On the ALLOCATED intrinsic function, see Section 9.4.10.

6.2.2. Allocation of Pointer Targets

When a pointer is allocated, the pointer is associated with a target and can be used to reference or define the target. (The target can be an array or a scalar, depending on how the pointer was declared).

Other pointers can become associated with the pointer target (or part of the pointer target) by pointer assignment.

In contrast to allocatable arrays, a pointer can be allocated a new target even if it is currently associated with a target. The previous association is broken and the pointer is then associated with the new target.

If the previous target was created by allocation, it becomes inaccessible unless it can still be referred to by other pointers that are currently associated with it.

The intrinsic function ASSOCIATED can be used to determine whether a pointer is currently associated with a target. (The association status of the pointer must be **defined**). For example:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR(:)
PTR => TAR
...
IF (ASSOCIATED(PTR,TAR))...
```

For More Information:

- On pointers, see Section 5.15.
- On pointer assignment, see Section 4.2.3.
- On the ASSOCIATED intrinsic function, see Section 9.4.15.

6.3. DEALLOCATE Statement

The DEALLOCATE statement frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated). It takes the following form:

```
DEALLOCATE (object [,object]...[,STAT=sv])
```

`object`

Is a structure component or the name of a variable, and must be a pointer or allocatable array.

`sv`

Is a scalar integer variable in which the status of the deallocation is stored.

Rules and Behavior

If a STAT variable is specified, it must not be deallocated in the DEALLOCATE statement in which it appears. If the deallocation is successful, the variable is set to zero. If the deallocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error). If no STAT variable is specified and an error condition occurs, program execution terminates.

It is recommended that all explicitly allocated storage be explicitly deallocated when it is no longer needed.

Examples

The following example shows deallocation of an allocatable array:

```
INTEGER ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE (A(10), B(-2:8,1:5))
...
DEALLOCATE(A, B, STAT = ALLOC_ERR)
```

For More Information:

On run-time error messages, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/] or online documentation.

6.3.1. Deallocation of Allocatable Arrays

If the DEALLOCATE statement specifies an array that is not currently allocated, an error occurs.

If an allocatable array with the TARGET attribute is deallocated, the association status of any pointer associated with it becomes undefined.

If a RETURN or END statement terminates a procedure, an allocatable array has one of the following allocation statuses:

- It keeps its previous allocation and association status if the following is true:
  - It has the SAVE attribute.
  - It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
  - It is accessible by host association.
- It remains allocated if it is accessed by use association.
- Otherwise, its allocation status is deallocated.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
SUBROUTINE TEST
  REAL, ALLOCATABLE, SAVE :: F(:,:)
  REAL, ALLOCATABLE :: E(:,:,:)
  ...
  IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7,14))
END SUBROUTINE TEST
```

Note that when subroutine TEST is exited, the allocation status of F is maintained because F has the SAVE attribute. Since E does not have the SAVE attribute, it is deallocated. On the next invocation of TEST, E will have the status of “not currently allocated.”

For More Information:

- On host association, see Section 15.5.1.2.
- On the TARGET attribute, see Section 5.18.
- On the RETURN statement, see Section 7.10.

- On the END statement, see Section 7.7.
- On the SAVE attribute, see Section 5.17.

6.3.2. Deallocation of Pointer Targets

A pointer must not be deallocated unless it has a defined association status. If the DEALLOCATE statement specifies a pointer that has undefined association status, or a pointer whose target was not created by allocation, an error occurs.

A pointer must not be deallocated if it is associated with an allocatable array, or it is associated with a portion of an object (such as an array element or an array section).

If a pointer is deallocated, the association status of any other pointer associated with the target (or portion of the target) becomes undefined.

Execution of a RETURN or END statement in a subprogram causes the pointer association status of any pointer declared (or accessed) in the procedure to become undefined, unless any of the following applies to the pointer:

- It has the SAVE attribute.
- It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
- It is accessible by host association.
- It is in blank common.
- It is in a named common block that appears in another scoping unit that is currently executing.
- It is the return value of a function declared with the POINTER attribute.

If the association status of a pointer becomes undefined, it cannot subsequently be referenced or defined.

Examples

The following example shows deallocation of a pointer:

```
INTEGER ERR
REAL, POINTER :: PTR_A(:)
...
ALLOCATE (PTR_A(10), STAT=ERR)
...
DEALLOCATE(PTR_A)
```

For More Information:

- On pointers, see Section 5.15.
- On host association, see Section 15.5.1.2.
- On the RETURN statement, see Section 7.10.
- On the END statement, see Section 7.7.

- On the SAVE attribute, see Section 5.17.
- On common blocks, see Section 5.4.
- On the NULL intrinsic function, which can be used to disassociate a pointer, see Section 9.4.110.

6.4. NULLIFY Statement

The NULLIFY statement disassociates a pointer from its target. It takes the following form:

```
NULLIFY (pointer-object [,pointer-object]...)
```

`pointer-object`

Is a structure component or the name of a variable; it must be a pointer (have the POINTER attribute).

Rules and Behavior

The initial association status of a pointer is undefined. You can use NULLIFY to initialize an undefined pointer, giving it disassociated status. Then the pointer can be tested using the intrinsic function ASSOCIATED.

Examples

The following is an example of the NULLIFY statement:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR_A(:), PTR_B(:)
PTR_A => TAR
PTR_B => TAR
...
NULLIFY(PTR_A)
```

After these statements are executed, PTR_A will have disassociated status, while PTR_B will continue to be associated with variable TAR.

For More Information:

- On the POINTER attribute, see Section 5.15.
- On pointer assignment, see Section 4.2.3.
- On the ASSOCIATED intrinsic function, see Section 9.4.15.
- On the NULL intrinsic function, which can also be used to disassociate a pointer, see Section 9.4.110.

Chapter 7. Execution Control

7.1. Overview

A program normally executes statements in the order in which they are written. Executable control constructs and statements modify this normal execution by transferring control to another statement in the program, or by selecting blocks (groups) of constructs and statements for execution or repetition.

In Fortran 95/90, control constructs (CASE, DO, and IF) can be named. The name must be a unique identifier in the scoping unit, and must appear on the initial line and terminal line of the construct. On the initial line, the name is separated from the statement keyword by a colon (:).

A block can contain any executable Fortran statement except an END statement. You can transfer control out of a block, but you cannot transfer control into another block.

DO loops cannot partially overlap blocks. The DO statement and its terminal statement must appear together in a statement block.

7.2. Branch Statements

Branching affects the normal execution sequence by transferring control to a labeled statement in the same scoping unit. The transfer statement is called the **branch statement**, while the statement to which the transfer is made is called the **branch target statement**.

Any executable statement can be a branch target statement, except for the following:

- CASE statement
- ELSE statement
- ELSE IF statement

Certain restrictions apply to the following statements:

Statement	Restriction
DO terminal statement	The branch must be taken from within its nonblock DO construct.[1]
END DO	The branch must be taken from within its block DO construct.
END IF	The branch should be taken from within its IF construct.[2]
END SELECT	The branch must be taken from within its CASE construct.

[1]If the terminal statement is shared by more than one nonblock DO construct, the branch can only be taken from within the innermost DO construct.

[2]You can branch to an END IF statement from outside the IF construct; this is a deleted feature in Fortran 95. VSI Fortran fully supports features deleted in Fortran 95.

The following branch statements are described in this section:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO (the ASSIGN statement is also described here)

- Arithmetic IF

For More Information:

- On IF constructs, see Section 7.8.
- On CASE constructs, see Section 7.4.
- On DO constructs, see Section 7.6.

7.2.1. Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same branch target statement every time it executes. It takes the following form:

```
GO TO label
```

`label`

Is the label of a valid branch target statement in the same scoping unit as the GO TO statement.

The unconditional GO TO statement transfers control to the branch target statement identified by the specified label.

The following are examples of GO TO statements:

```
GO TO 7734 GO TO 99999
```

7.2.2. Computed GO TO Statement

The computed GO TO statement transfers control to one of a set of labeled branch target statements based on the value of an expression. It is an obsolescent feature in Fortran 95.

The computed GO TO statement takes the following form:

```
GO TO (label-list)[,] expr
```

`label-list`

Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the computed GO TO statement. (Also called the transfer list.) The same label can appear more than once in this list.

`expr`

Is a scalar numeric expression in the range 1 to n, where *n* is the number of statement labels in *label-list*. If necessary, it is converted to integer data type.

Rules and Behavior

When the computed GO TO statement is executed, the expression is evaluated first. The value of the expression represents the ordinal position of a label in the associated list of labels. Control is transferred to the statement identified by the label. For example, if the list contains (30,20,30,40) and the value of the expression is 2, control is transferred to the statement identified with label 20.

If the value of the expression is less than 1 or greater than the number of labels in the list, control is transferred to the next executable statement or construct following the computed GO TO statement.

Examples

The following example shows valid computed GO TO statements:

```
GO TO (12,24,36), INDEX
GO TO (320,330,340,350,360), SITU(J,K) + 1
```

7.2.3. ASSIGN and Assigned GO TO Statements

The ASSIGN statement assigns a label to an integer variable. Subsequently, this variable can be used as a branch target statement by an assigned GO TO statement or as a format specifier in a formatted input/output statement.

The ASSIGN and assigned GO TO statements have been deleted in Fortran 95; they were obsolescent features in Fortran 90. VSI Fortran fully supports features deleted in Fortran 95.

For More Information:

On obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95, see Appendix A.

7.2.3.1. ASSIGN Statement

The ASSIGN statement assigns a statement label value to an integer variable. It takes the following form:

```
ASSIGN label TO var
```

`label`

Is the label of a branch target or FORMAT statement in the same scoping unit as the ASSIGN statement.

`var`

Is a scalar integer variable on OpenVMS Alpha and I64 platforms and an INTEGER*8 on OpenVMS x86-64 platforms.

Rules and Behavior

When an ASSIGN statement is executed, the statement label is assigned to the integer variable. The variable is then undefined as an integer variable and can only be used as a label (unless it is later redefined with an integer value).

The ASSIGN statement must be executed before the statements in which the assigned variable is used.

Examples

The following example shows ASSIGN statements:

```
INTEGER ERROR
...
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
ASSIGN 250 TO ERROR
```

On OpenVMS IA64 and Alpha, NSTART and KSTOP are integer variables implicitly, but ERROR must be previously declared as an integer variable. On x86-64, NSTART, KSTOP, and ERROR must be INTEGER*8.

The following statement associates the variable NUMBER with the statement label 100:

```
 ASSIGN 100 TO NUMBER
```

If an arithmetic operation is subsequently performed on variable NUMBER (such as follows), the run-time behavior is unpredictable:

```
NUMBER = NUMBER + 1
```

To return NUMBER to the status of an integer variable, you can use the following statement:

```
NUMBER = 10
```

This statement dissociates NUMBER from statement 100 and assigns it an integer value of 10. Once NUMBER is returned to its integer variable status, it can no longer be used in an assigned GO TO statement.

7.2.3.2. Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to a variable. The assigned GO TO statement takes the following form:

```
GO TO var [[,] (label-list)]
```

var

Is a scalar integer variable on OpenVMS Alpha and I64 platforms and an INTEGER*8 on OpenVMS x86-64 platforms.

label-list

Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the assigned GO TO statement. The same label can appear more than once in this list.

Rules and Behavior

The variable must have a statement label value assigned to it by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

If a list of labels appears, the statement label assigned to the variable must be one of the labels in the list.

Both the assigned GO TO statement and its associated ASSIGN statement must be in the same scoping unit.

Examples

The following example is equivalent to `GO TO 200`:

```
ASSIGN 200 TO IGO
GO TO IGO
```

The following example is equivalent to `GO TO 450`:

```
ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
```

The following example shows an invalid use of an assigned variable:

```
ASSIGN 10 TO I
J = I
GO TO J
```

In this case, variable J is not the variable assigned to, so it cannot be used in the assigned GO TO statement.

7.2.4. Arithmetic IF Statement

The arithmetic IF statement conditionally transfers control to one of three statements, based on the value of an arithmetic expression. It is an obsolescent feature in Fortran 95 and Fortran 90.

The arithmetic IF statement takes the following form:

```
IF (expr) label1, label2, label3
```

expr

Is a scalar numeric expression of type integer or real (enclosed in parentheses).

label1, label2, label3

Are the labels of valid branch target statements that are in the same scoping unit as the arithmetic IF statement.

Rules and Behavior

All three labels are required, but they do not need to refer to three different statements. The same label can appear more than once in the same arithmetic IF statement.

During execution, the expression is evaluated first. Depending on the value of the expression, control is then transferred as follows:

If the Value of *expr* is:	**Control Transfers To:**
Less than 0	Statement *label1*
Equal to 0	Statement *label2*
Greater than 0	Statement *label3*

Examples

The following example transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (THETA-CHI) 50,50,100
```

The following example transfers control to statement 40 if the value of the integer variable NUMBER is even. It transfers control to statement 20 if the value is odd.

```
IF (NUMBER / 2*2 - NUMBER) 20,40,20
```

For More Information:

On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

7.3. CALL Statement

The CALL statement transfers control to a subroutine subprogram. It takes the following form:

```
CALL sub [([a-arg [,a-arg]...]) ]
```

`sub`

Is the name of the subroutine subprogram or other external procedure, or a dummy argument associated with a subroutine subprogram or other external procedure.

`a-arg`

Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the subroutine. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, the name of a procedure, or an alternate return specifier. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

An alternate return specifier is an asterisk (*), or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90).

Rules and Behavior

When the CALL statement is executed, any expressions in the actual argument list are evaluated, then control is passed to the first executable statement or construct in the subroutine. When the subroutine

finishes executing, control returns to the next executable statement following the CALL statement, or to a statement identified by an alternate return label (if any).

If an argument list appears, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Table 9.1).

You can use a CALL statement to invoke a function as long as the function is not one of the following types:

- REAL(8)
- REAL(16)
- COMPLEX(8)
- COMPLEX(16)
- CHARACTER

Examples

The following example shows valid CALL statements:

```
CALL CURVE(BASE,3.14159+X,Y,LIMIT,R(LT+2))

CALL PNTOUT(A,N,'ABCD')

CALL EXIT

CALL MULT(A,B,*10,*20,C)       ! The asterisks and ampersands denote
CALL SUBA(X,&30,&50,Y)         !     alternate returns
```

The following example shows a subroutine with argument keywords:

```
PROGRAM KEYWORD_EXAMPLE
  INTERFACE
    SUBROUTINE TEST_C(I, L, J, KYWD2, D, F, KYWD1)
      INTEGER I, L(20), J, KYWD1
      REAL, OPTIONAL :: D, F
      COMPLEX KYWD2
      ...
    END SUBROUTINE TEST_C
  END INTERFACE
  INTEGER I, J, K
  INTEGER L(20)
  COMPLEX Z1
  CALL TEST_C(I, L, J, KYWD1 = K, KYWD2 = Z1)
```

```
...
```

The first three actual arguments are associated with their corresponding dummy arguments by position. The argument keywords are associated by keyword name, so they can appear in any order.

Note that the interface to subroutine TEST has two optional arguments that have been omitted in the CALL statement.

The following is another example of a subroutine call with argument keywords:

```
CALL TEST(X, Y, N, EQUALITIES = Q, XSTART = X0)
```

The first three arguments are associated by position.

For More Information:

- On procedure arguments, see Section 8.8.
- On subroutines, see Section 8.5.3.
- On optional arguments, see Section 5.13.
- On dummy arguments, see Section 8.8.7.
- On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

7.4. CASE Construct

The CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement.

The CASE construct takes the following form:

```
[name:] SELECT CASE (expr)
[CASE (case-value [,case-value]...) [name]
  block]...
[CASE DEFAULT [name]
  block]
END SELECT [name]
```

name

Is the name of the CASE construct.

expr

Is a scalar expression of type integer, logical, or character (enclosed in parentheses). Evaluation of this expression results in a value called the *case index*.

case-value

Is one or more scalar integer, logical, or character initialization expressions enclosed in parentheses. Each *case-value* must be of the same type and kind parameter as *expr*. If the type is character, *case-value* and *expr* can be of different lengths, but their kind parameter must be the same.

Integer and character expressions can be expressed as a range of case values, taking one of the following forms:

```
low:high
low:
:high
```

Case values must not overlap.

`block`

Is a sequence of zero or more statements or constructs.

Rules and Behavior

If a construct name is specified in a SELECT CASE statement, the same name must appear in the corresponding END SELECT statement. The same construct name can optionally appear in any CASE statement in the construct. The same construct name must not be used for different named constructs in the same scoping unit.

The case expression (*expr*) is evaluated first. The resulting case index is compared to the case values to find a matching value (there can only be one). When a match occurs, the block following the matching case value is executed and the construct terminates.

The following rules determine whether a match occurs:

- When the case value is a single value (no colon appears), a match occurs as follows:

Data Type	**A Match Occurs If:**
Logical	case-index .EQV. case-value
Integer or Character	case-index == case-value

- When the case value is a range of values (a colon appears), a match depends on the range specified, as follows:

Range	**A Match Occurs If:**
low:	case-index >= low
:high	case-index <= high
low:high	low <= case-index <= high

The following are all valid case values:

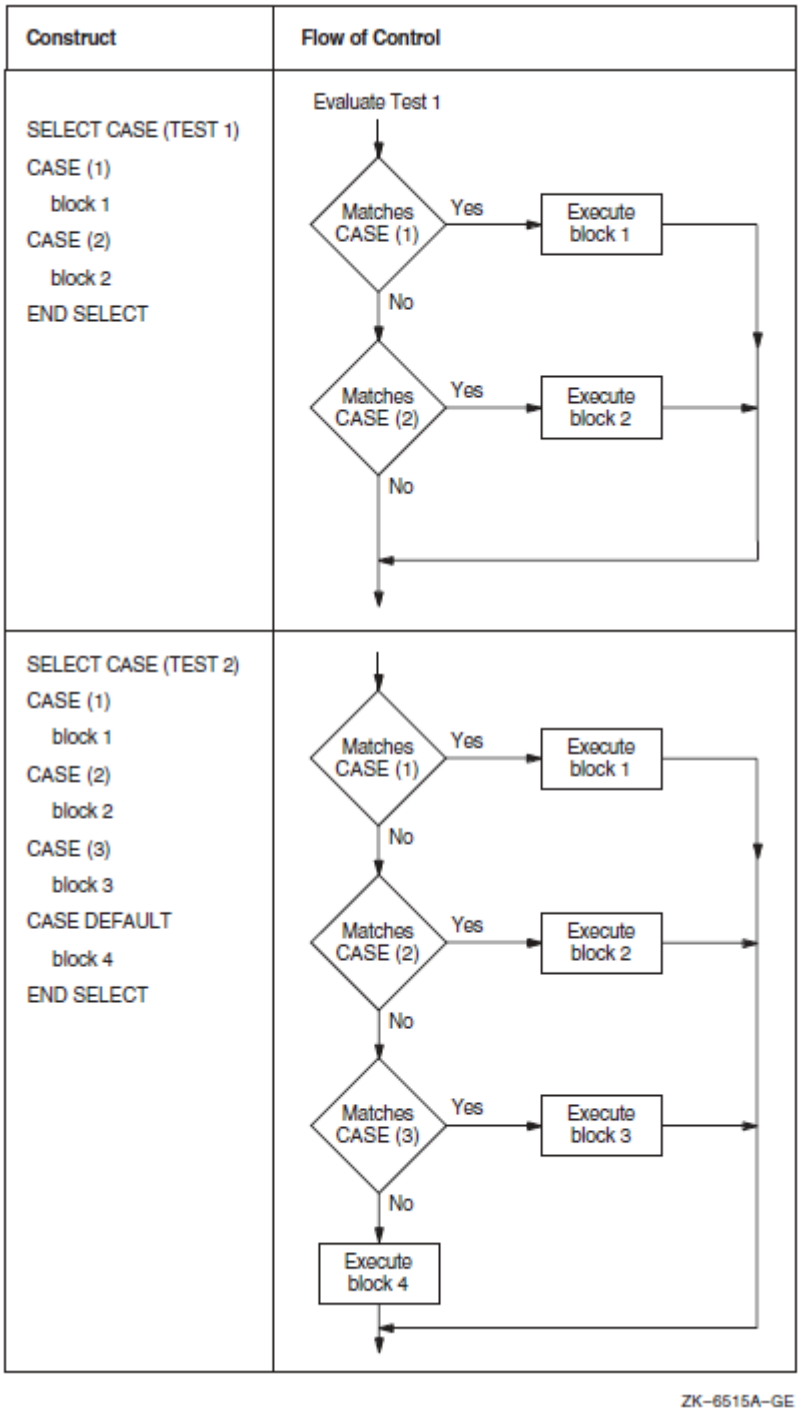
```
CASE (1, 4, 7, 11:14, 22)      ! Individual values as specified:
                               !     1, 4, 7, 11, 12, 13, 14, 22
CASE (:-1)                     ! All values less than zero
CASE (0)                       ! Only zero
CASE (1:)                      ! All values above zero
```

If no match occurs but a CASE DEFAULT statement is present, the block following that statement is executed and the construct terminates.

If no match occurs and no CASE DEFAULT statement is present, no block is executed, the construct terminates, and control passes to the next executable statement or construct following the END SELECT statement.

Figure 7.1 shows the flow of control in a CASE construct.

Figure 7.1. Flow of Control in CASE Constructs



You cannot use branching statements to transfer control to a CASE statement. However, branching to a SELECT CASE statement is allowed. Branching to the END SELECT statement is allowed only from within the CASE construct.

Examples

The following are examples of CASE constructs:

```
INTEGER FUNCTION STATUS_CODE (I)
  INTEGER I
```

```
  CHECK_STATUS: SELECT CASE (I)
  CASE (:-1)
    STATUS_CODE = -1
  CASE (0)
    STATUS_CODE = 0
  CASE (1:)
    STATUS_CODE = 1
  END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE

SELECT CASE (J)
CASE (1, 3:7, 9)    ! Values: 1, 3, 4, 5, 6, 7, 9
  CALL SUB_A
CASE DEFAULT
  CALL SUB_B
END SELECT
```

The following three examples are equivalent:

```
1. SELECT CASE (ITEST .EQ. 1)
   CASE (.TRUE.)
     CALL SUB1 ()
   CASE (.FALSE.)
     CALL SUB2 ()
   END SELECT

2. SELECT CASE (ITEST)
   CASE DEFAULT
     CALL SUB2 ()
   CASE (1)
     CALL SUB1 ()
   END SELECT

3. IF (ITEST .EQ. 1) THEN
     CALL SUB1 ()
   ELSE
     CALL SUB2 ()
   END IF
```

7.5. CONTINUE Statement

The CONTINUE statement is primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.

The CONTINUE statement takes the following form:

```
CONTINUE
```

The statement by itself does nothing and has no effect on program results or execution sequence.

The following example shows a CONTINUE statement:

```
    DO 150 I = 1,40
40  Y = Y + 1
    Z = COS(Y)
    PRINT *, Z
    IF (Y .LT. 30) GO TO 150
    GO TO 40
```

```
150 CONTINUE
```

7.6. DO Constructs

The DO construct controls the repeated execution of a block of statements or constructs. (This repeated execution is called a **loop**).

The number of iterations of a loop can be specified in the initial DO statement in the construct, or the number of iterations can be left indefinite by a simple DO ("DO forever") construct or DO WHILE statement.

The EXIT and CYCLE statements modify the execution of a loop. An EXIT statement terminates execution of a loop, while a CYCLE statement terminates execution of the current iteration of a loop. For example:

```
DO
  READ (EUNIT, IOSTAT=IOS) Y
  IF (IOS /= 0) EXIT
  IF (Y < 0) CYCLE
  CALL SUB_A(Y)
END DO
```

If an error or end-of-file occurs, the DO construct terminates. If a negative value for Y is read, the program skips to the next READ statement.

For More Information:

- On the CYCLE statement, see Section 7.6.4.
- On the EXIT statement, see Section 7.6.5.
- On DO loops in FORALL constructs, see Section 4.2.5.

7.6.1. Forms for DO Constructs

A DO construct takes one of the following forms:

```
[name:] DO [label][,] [loop-control]
   block
[label] term-stmt

DO label[,] [loop-control]
```

`name`

Is the name of the DO construct.

`label`

Is a statement label identifying the terminal statement.

`loop-control`

Is a DO iteration (see Section 7.6.2.1) or a (DO) WHILE statement (see Section 7.6.3).

block

Is a sequence of zero or more statements or constructs.

term-stmt

Is the terminal statement for the construct.

Rules and Behavior

A block DO construct is terminated by an END DO or CONTINUE statement. If the block DO statement contains a label, the terminal statement must be identified with the same label. If no label appears, the terminal statement must be an END DO statement.

If a construct name is specified in a block DO statement, the same name must appear in the terminal END DO statement. If no construct name is specified in the block DO statement, no name can appear in the terminal END DO statement.

A nonblock DO construct is terminated by an executable statement (or construct) that is identified by the label specified in the nonblock DO statement. A nonblock DO construct can share a terminal statement with another nonblock DO construct. A block DO construct cannot share a terminal statement.

The following cannot be terminal statements for nonblock DO constructs:

- CONTINUE (allowed if it is a shared terminal statement)
- CYCLE
- END (for a program or subprogram)
- EXIT
- GO TO (unconditional or assigned)
- Arithmetic IF
- RETURN
- STOP

The nonblock DO construct is an obsolescent feature in Fortran 95 and Fortran 90.

Examples

The following example shows equivalent block DO and nonblock DO constructs:

```
   DO I = 1, N                 ! Block DO
     TOTAL = TOTAL + B(I)
   END DO

   DO 20 I = 1, N              ! Nonblock DO
20 TOTAL = TOTAL + B(I)
```

The following example shows a simple block DO construct (contains no iteration count or DO WHILE statement):

```
DO
  READ *, N
  IF (N == 0) STOP
  CALL SUBN
```

```
END DO
```

The DO block executes repeatedly until the value of zero is read. Then the DO construct terminates.

The following example shows a named block DO construct:

```
LOOP_1: DO I = 1, N
          A(I) = C * B(I)
        END DO LOOP_1
```

The following example shows a nonblock DO construct with a shared terminal statement:

```
   DO 20 I = 1, N
   DO 20 J = 1 + I, N
20 RESULT(I,J) = 1.0 / REAL(I + J)
```

For More Information:

On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

7.6.2. Execution of DO Constructs

The range of a DO construct includes all the statements and constructs that follow the DO statement, up to and including the terminal statement. If the DO construct contains another construct, the inner (nested) construct must be entirely contained within the DO construct.

Execution of a DO construct differs depending on how the loop is controlled, as follows:

- For simple DO constructs, there is no loop control. Statements in the DO range are repeated until the DO statement is terminated explicitly by a statement within the range.
- For iterative DO statements, loop control is specified as *do-var = expr1, expr2 [,expr3]*. An iteration count specifies the number of times the DO range is executed. (For more information on iteration loop control, see Section 7.6.2.1).
- For DO WHILE statements, loop control is specified as a DO range. The DO range is repeated as long as a specified condition remains true. Once the condition is evaluated as false, the DO construct terminates. (For more information on the DO WHILE statement, see Section 7.6.3).

7.6.2.1. Iteration Loop Control

DO iteration loop control takes the following form:

```
do-var = expr1, expr2 [,expr3]
```

do-var

Is the name of a scalar variable of type integer or real. It cannot be the name of an array element or structure component.

expr

Is a scalar numeric expression of type integer or real. If it is not the same type as *do-var*, it is converted to that type.

Rules and Behavior

A DO variable or expression of type real is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. VSI Fortran fully supports features deleted in Fortran 95.

The following steps are performed in iteration loop control:

1. The expressions *expr1*, *expr2*, and *expr3* are evaluated to respectively determine the initial, terminal, and increment parameters.

   The increment parameter (*expr3*) is optional and must not be zero. If an increment parameter is not specified, it is assumed to be of type default integer with a value of 1.

2. The DO variable (*do-var*) becomes defined with the value of the initial parameter (*expr1*).

3. The iteration count is determined as follows:

   MAX(INT((expr2 − expr1 + expr3) ÷ expr3), 0)

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

   expr1 > expr2 and expr3 > 0
   expr1 < expr2 and expr3 < 0

4. The iteration count is tested. If the iteration count is zero, the loop terminates and the DO construct becomes inactive. (A compiler option can affect this; see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/] for more information.) If the iteration count is nonzero, the range of the loop is executed.

5. The iteration count is decremented by one, and the DO variable is incremented by the value of the increment parameter, if any.

After termination, the DO variable retains its last value (the one it had when the iteration count was tested and found to be zero).

The DO variable must not be redefined or become undefined during execution of the DO range.

If you change variables in the initial, terminal, or increment expressions during execution of the DO construct, it does not affect the iteration count. The iteration count is fixed each time the DO construct is entered.

Examples

The following example specifies 25 iterations:

```
DO 100 K=1,50,2
```

K=49 during the final iteration, K=51 after the loop.

The following example specifies 27 iterations:

```
DO 350 J=50,-2,-2
```

J=–2 during the final iteration, J=–4 after the loop.

The following example specifies 9 iterations:

```
DO NUMBER=5,40,4
```

NUMBER=37 during the final iteration, NUMBER=41 after the loop. The terminating statement of this DO loop must be END DO.

For More Information:

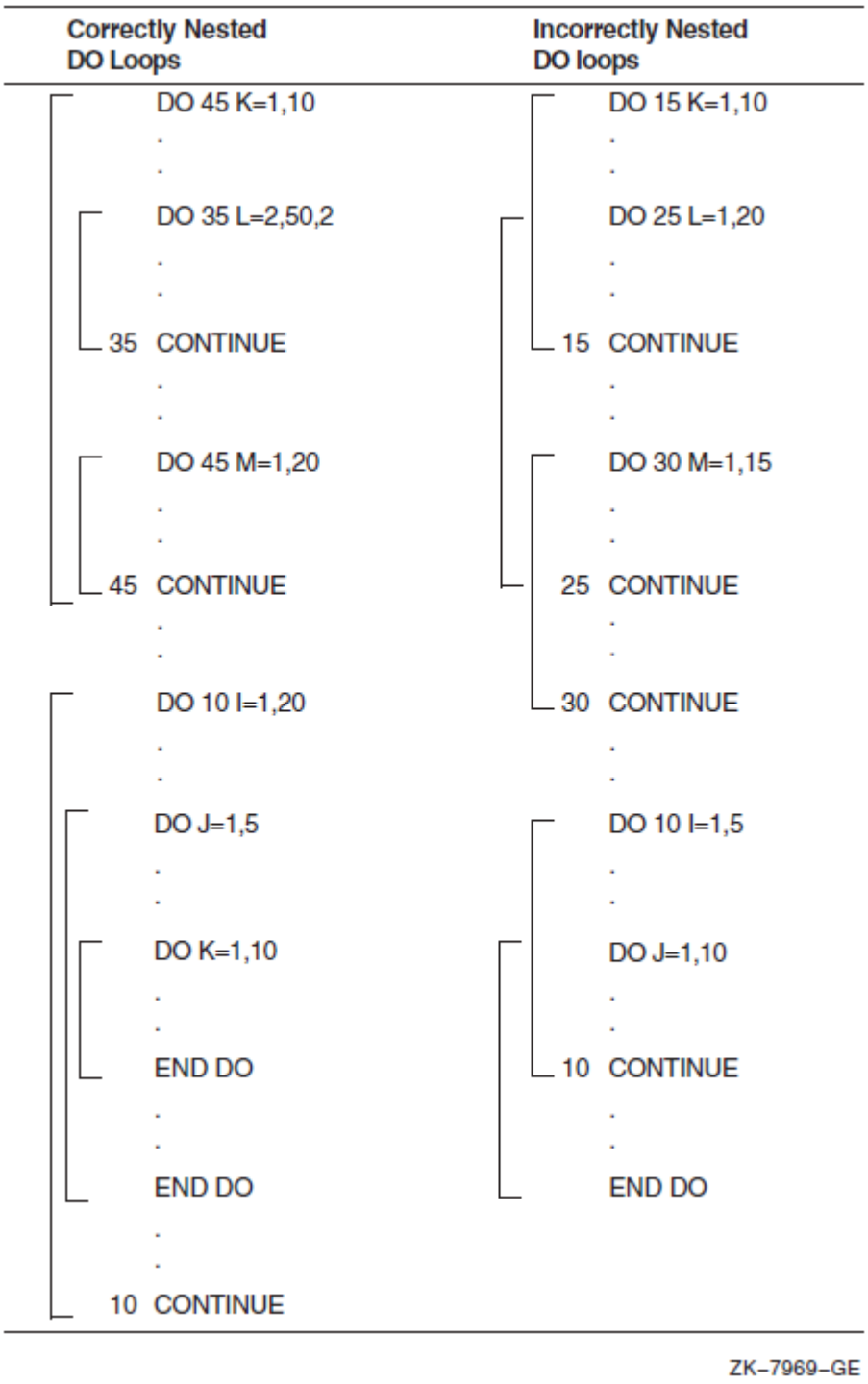
On obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95, see Appendix A.

7.6.2.2. Nested DO Constructs

A DO construct can contain one or more complete DO constructs (loops). The range of an inner nested DO construct must lie completely within the range of the next outer DO construct. Nested nonblock DO constructs can share a labeled terminal statement.

Figure 7.2 shows correctly and incorrectly nested DO constructs.

Figure 7.2. Nested DO Constructs



In a nested DO construct, you can transfer control from an inner construct to an outer construct. However, you cannot transfer control from an outer construct to an inner construct.

If two or more nested DO constructs share the same terminal statement, you can transfer control to that statement only from within the range of the innermost construct. Any other transfer to that statement constitutes a transfer from an outer construct to an inner construct, because the shared statement is part of the range of the innermost construct.

7.6.2.3. Extended Range

A DO construct has an extended range if both of the following are true:

- The DO construct contains a control statement that transfers control out of the construct.
- Another control statement returns control back into the construct after execution of one or more statements.

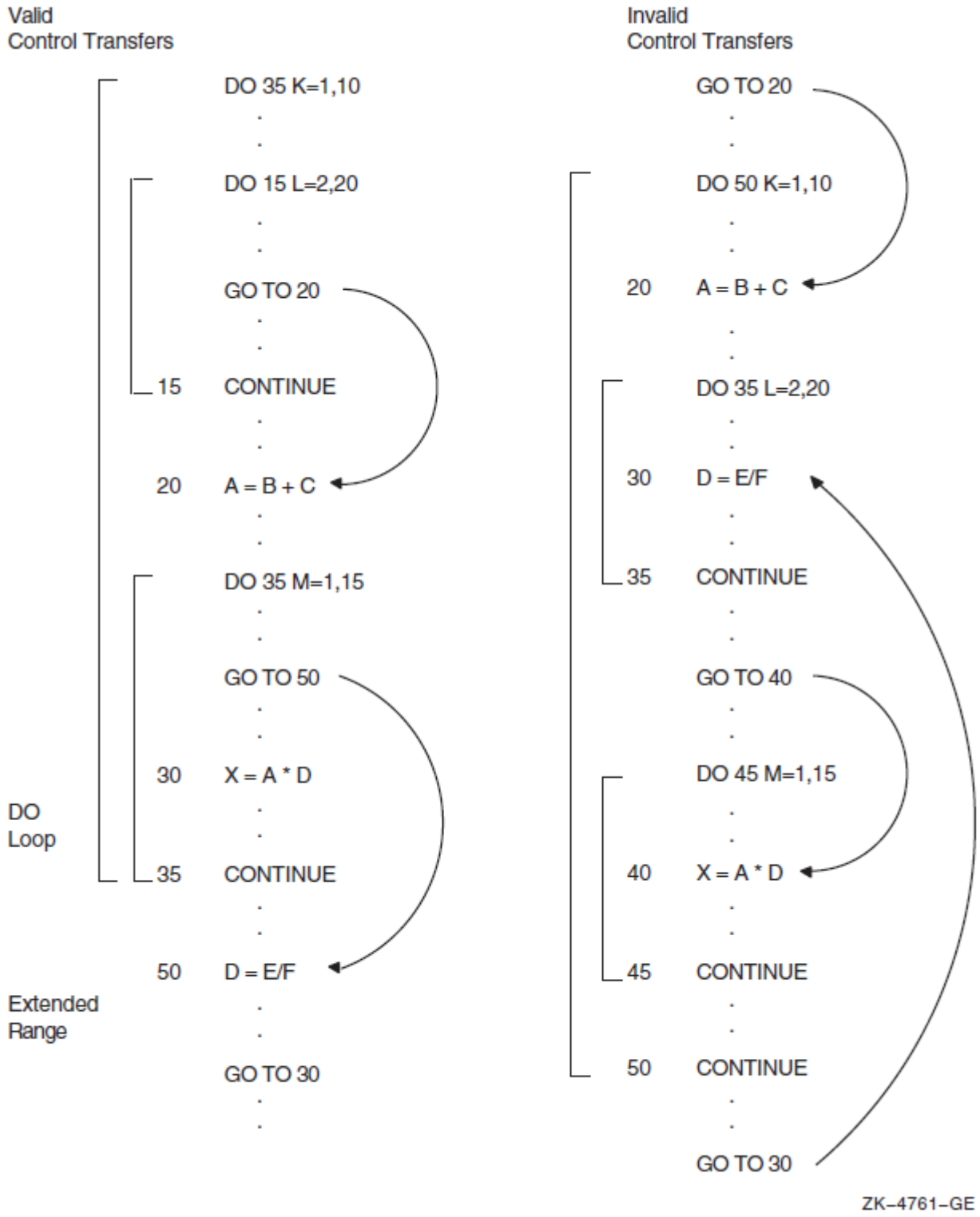
The range of the construct is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the construct.

The following rules apply to a DO construct with extended range:

- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.
- The extended range of a DO statement must not change the control variable of the DO statement.

Figure 7.3 illustrates valid and invalid extended range control transfers.

Figure 7.3. Control Transfers and Extended Range



7.6.3. DO WHILE Statement

The DO WHILE statement executes the range of a DO construct while a specified condition remains true. The statement takes the following form:

```
DO [label][,] WHILE (expr)
```

label

Is a label specifying an executable statement in the same program unit.

expr

Is a scalar logical expression enclosed in parentheses.

Rules and Behavior

Before each execution of the DO range, the logical expression is evaluated. If it is true, the statements in the body of the loop are executed. If it is false, the DO construct terminates and control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement.

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

Examples

The following example shows a DO WHILE statement:

```
CHARACTER*132 LINE
...
I = 1
DO WHILE (LINE(I:I) .EQ. ' ')
  I = I + 1
END DO
```

The following examples show required and optional END DO statements:

Required	Optional

`DO WHILE (I .GT. J)`
`  ARRAY(I,J) = 1.0`
`  I = I - 1`
`END DO`
	`DO 10 WHILE (I .GT. J)`
	`  ARRAY(I,J) = 1.0`
	`  I = I - 1`
	`10 END DO`

7.6.4. CYCLE Statement

The CYCLE statement interrupts the current execution cycle of the innermost (or named) DO construct.

The CYCLE statement takes the following form:

```
CYCLE [name]
```

name

Is the name of the DO construct.

Rules and Behavior

When a CYCLE statement is executed, the following occurs:

1. The current execution cycle of the named (or innermost) DO construct is terminated.

   If a DO construct name is specified, the CYCLE statement must be within the range of that construct.

2. The iteration count (if any) is decremented by 1.

3. The DO variable (if any) is incremented by the value of the increment parameter (if any).

4. A new iteration cycle of the DO construct begins.

Any executable statements following the CYCLE statement (including a labeled terminal statement) are not executed.

A CYCLE statement can be labeled, but it cannot be used to terminate a DO construct.

Examples

The following example shows a CYCLE statement:

```
DO I =1, 10
  A(I) = C + D(I)
  IF (D(I) < 0) CYCLE    ! If true, the next statement is omitted
  A(I) = 0               ! from the loop and the loop is tested again.
END DO
```

7.6.5. EXIT Statement

The EXIT statement terminates execution of a DO construct. It takes the following form:

```
EXIT [name]
```

name

Is the name of the DO construct.

Rules and Behavior

The EXIT statement causes execution of the named (or innermost) DO construct to be terminated.

If a DO construct name is specified, the EXIT statement must be within the range of that construct.

Any DO variable present retains its last defined value.

An EXIT statement can be labeled, but it cannot be used to terminate a DO construct.

Examples

The following example shows an EXIT statement:

```
LOOP_A : DO I = 1, 15
  N = N + 1
  IF (N > I) EXIT LOOP_A
END DO LOOP_A
```

7.7. END Statement

The END statement marks the end of a program unit. It takes one of the following forms:

```
END [PROGRAM [program-name]]
END [FUNCTION [function-name]]
END [SUBROUTINE [subroutine-name]]
END [MODULE [module-name]]
END [BLOCK DATA [block-data-name]]
```

For internal procedures and module procedures, you must specify the FUNCTION and SUBROUTINE keywords in the END statement; otherwise, the keywords are optional.

In main programs, function subprograms, and subroutine subprograms, END statements are executable and can be branch target statements. If control reaches the END statement in these program units, the following occurs:

- In a main program, execution of the program terminates.
- In a function or subroutine subprogram, a RETURN statement is implicitly executed.

The END statement cannot be continued in a program unit, and no other statement in the program unit can have an initial line that appears to be the program unit END statement.

The END statements in a module or block data program unit are nonexecutable.

For More Information:

- On program units and procedures, see Chapter 8.
- On branch target statements, see Section 7.2.

7.8. IF Construct and Statement

The IF construct conditionally executes one block of statements or constructs.

The IF statement conditionally executes one statement.

The decision to transfer control or to execute the statement or block is based on the evaluation of a logical expression within the IF statement or construct.

For More Information:

On the arithmetic IF statement, see Section 7.2.4.

7.8.1. IF Construct

The IF construct conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. (This construct was called a block IF statement in FORTRAN 77).

The IF construct takes the following form:

```
[name:] IF (expr) THEN
   block
[ELSE IF (expr) THEN [name]
   block]...
[ELSE [name]
   block]
END IF [name]
```

name

Is the name of the IF construct.

expr

Is a scalar logical expression enclosed in parentheses.

block

Is a sequence of zero or more statements or constructs.

Rules and Behavior

If a construct name is specified at the beginning of an IF THEN statement, the same name must appear in the corresponding END IF statement. The same construct name must not be used for different named constructs in the same scoping unit.

Depending on the evaluation of the logical expression, one block or no block is executed. The logical expressions are evaluated in the order in which they appear, until a true value is found or an ELSE or END IF statement is encountered.

Once a true value is found or an ELSE statement is encountered, the block immediately following it is executed and the construct execution terminates.

If none of the logical expressions evaluate to true and no ELSE statement appears in the construct, no block in the construct is executed and the construct execution terminates.

Note

No additional statement can be placed after the IF THEN statement in a block IF construct. For example, the following statement is invalid in the block IF construct:

```
IF (e) THEN I = J
```

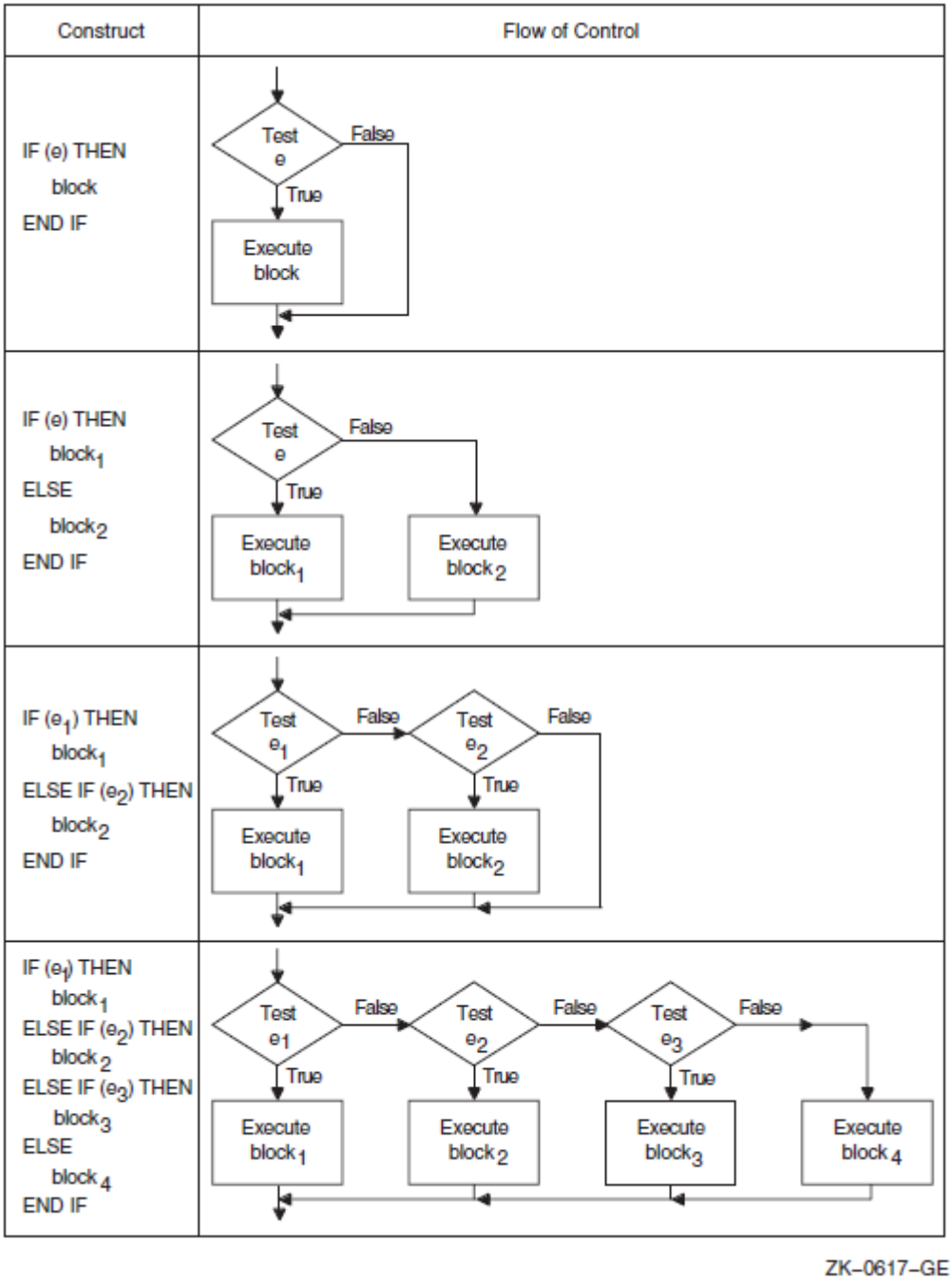
This statement is translated as the following logical IF statement:

```
IF (e) THENI = J
```

You cannot use branching statements to transfer control to an ELSE IF statement or ELSE statement. However, you can branch to an END IF statement from within the IF construct.

Figure 7.4 shows the flow of control in IF constructs.

Figure 7.4. Flow of Control in IF Constructs



ZK–0617–GE

You can include an IF construct in the statement block of another IF construct, if the nested IF construct is completely contained within a statement block. It cannot overlap statement blocks.

Examples

The following example shows the simplest form of an IF construct:

Form	Example

`IF (expr) THEN`
`  block`
	`IF (ABS(ADJU) .GE. 1.0E-6) THEN`
	`  TOTERR = TOTERR + ABS(ADJU)`
	`  QUEST = ADJU/FNDVAL`

Form	Example
END IF	END IF

This construct conditionally executes the block of statements between the IF THEN and the END IF statements.

The following example shows an IF construct containing an ELSE statement:

Form	Example

IF (expr) THEN
   block1
 	
ELSE
   block2
END IF
	IF (NAME .LT. 'N') THEN
	   IFRONT = IFRONT + 1
	   FRLET(IFRONT) = NAME(1:2)
	ELSE
	   IBACK = IBACK + 1
	END IF

Block1 consists of all the statements between the IF THEN and ELSE statements. Block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than 'N', block1 is executed. If the value of NAME is greater than or equal to 'N', block2 is executed.

The following example shows an IF construct containing an ELSE IF THEN statement:

Form	Example

IF (expr) THEN
   block1
 	
ELSE IF (expr) THEN
   block2
 	
END IF
	IF (A .GT. B) THEN
	   D = B
	   F = A - B
	ELSE IF (A .GT. B/2.) THEN
	   D = B/2.
	   F = A - B/2.
	END IF

If A is greater than B, block1 is executed. If A is not greater than B, but A is greater than B/2, block2 is executed. If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed. Control transfers directly to the next executable statement after the END IF statement.

The following example shows an IF construct containing several ELSE IF THEN statements and an ELSE statement:

Form	Example

IF (expr) THEN
   block1
 	
ELSE IF (expr) THEN
   block2
 	
ELSE IF (expr) THEN
   block3
 	
ELSE
   block4
 	
END IF
	IF (A .GT. B) THEN
	   D = B
	   F = A - B
	ELSE IF (A .GT. C) THEN
	   D = C
	   F = A - C
	ELSE IF (A .GT. Z) THEN
	   D = Z
	   F = A - Z
	ELSE
	   D = 0.0
	   F = A
	END IF

If A is greater than B, block1 is executed. If A is not greater than B but is greater than C, block2 is executed. If A is not greater than B or C but is greater than Z, block3 is executed. If A is not greater than B, C, or Z, block4 is executed.

The following example shows a nested IF construct:

Form	Example

`IF (expr1) THEN`
`  block1`
`  IF (expr2) THEN`
`  block1a`
`  ELSE`
`  block1b`
`  END IF`
`ELSE`
`  block2`
`END IF`
	`IF (A .LT. 100) THEN`
	`  INRAN = INRAN + 1`
	`  IF (ABS(A-AVG) .LE. 5.) THEN`
	`  INAVG = INAVG + 1`
	`  ELSE`
	`  OUTAVG = OUTAVG + 1`
	`  END IF`
	`ELSE`
	`  OUTRAN = OUTRAN + 1`
	`END IF`

If A is less than 100, the code immediately following the IF is executed. This code contains a nested IF construct. If the absolute value of A minus AVG is less than or equal to 5, block1a is executed. If the absolute value of A minus AVG is greater than 5, block1b is executed.

If A is greater than or equal to 100, block2 is executed, and the nested IF construct (in block1) is not executed.

The following example shows a named IF construct:

```
BLOCK_A: IF (D > 0.0) THEN        ! Initial statement for named construct

  RADIANS = ACOS(D)               ! These two statements
  DEGREES = ACOSD(D)              !       form a block

END IF BLOCK_A                    ! Terminal statement for named construct
```

7.8.2. IF Statement

The IF statement conditionally executes one statement based on the value of a logical expression. (This statement was called a logical IF statement in FORTRAN 77).

The IF statement takes the following form:

```
IF (expr) stmt
```

`expr`

Is a scalar logical expression enclosed in parentheses.

`stmt`

Is any complete, unlabeled, executable Fortran statement, except for the following:

- A CASE, DO, IF, FORALL, or WHERE construct
- Another IF statement
- The END statement for a program, function, or subroutine

When an IF statement is executed, the logical expression is evaluated first. If the value is true, the statement is executed. If the value is false, the statement is not executed and control transfers to the next statement in the program.

Examples

The following examples show valid IF statements:

```
IF (J.GT.4 .OR. J.LT.1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5D0)

IF (ENDRUN) CALL EXIT
```

7.9. PAUSE Statement

The PAUSE statement temporarily suspends program execution until the user or system resumes execution. The PAUSE statement is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. VSI Fortran fully supports features deleted in Fortran 95.

The PAUSE statement takes the following form:

```
PAUSE [pause-code]
```

pause-code

Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Fortran 90 and FORTRAN 77 limit digits to five.)

Rules and Behavior

If you specify *pause-code*, the PAUSE statement displays the specified message and then displays the default prompt.

If you do not specify *pause-code*, the system displays the following default message:

```
FORTRAN PAUSE
```

The system prompt is then displayed.

The effect of PAUSE differs depending on whether the program is an interactive or batch process, as follows:

- If a program is an interactive process, the program is suspended until you enter one of the following commands:
  - CONTINUE resumes execution at the next executable statement.
  - DEBUG resumes execution under control of the OpenVMS Debugger.
  - EXIT terminates execution.

In general, most other commands also terminate execution.

- If a program is a batch process, the program is not suspended. If you specify a value for *pause-code*, this value is written to SYS$OUTPUT.

Examples

The following examples show valid PAUSE statements:

```
PAUSE 701
PAUSE 'ERRONEOUS RESULT DETECTED'
```

For More Information:

On obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95, see Appendix A.

7.10. RETURN Statement

The RETURN statement transfers control from a subprogram to the calling program unit.

The RETURN statement takes the following form:

```
RETURN [expr]
```

expr

Is a scalar expression that is converted to an integer value if necessary.

The *expr* is only allowed in subroutines; it indicates an alternate return. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90).

Rules and Behavior

When a RETURN statement is executed in a function subprogram, control is transferred to the referencing statement in the calling program unit.

When a RETURN statement is executed in a subroutine subprogram, control is transferred to the first executable statement following the CALL statement that invoked the subroutine, or to the alternate return (if one is specified).

Examples

The following shows how alternate returns can be used in a subroutine:

```
   CALL CHECK(A, B, *10, *20, C)
   ...
10 ...
20 ...
   SUBROUTINE CHECK(X, Y, *, *, C)
   ...
50   IF (X) 60, 70, 80
60   RETURN
```

```
70   RETURN 1
80   RETURN 2
   END
```

The value of X determines the return, as follows:

- If X < 0, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the calling program.
- If X == 0, the first alternate return (RETURN 1) occurs and control is transferred to the statement identified with label 10.
- If X > 0, the second alternate return (RETURN 2) occurs and control is transferred to the statement identified with label 20.

Note that an asterisk (*) specifies the alternate return. An ampersand (&) can also specify an alternate return in a CALL statement, but not in a subroutine's dummy argument list.

For More Information:

- On the CALL statement, see Section 7.3.
- On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

7.11. STOP Statement

The STOP statement terminates program execution before the end of the program unit. It takes the following form:

```
STOP [stop-code]
```

stop-code

Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Fortran 95/90 and FORTRAN 77 limit digits to five).

If you specify stop-code, the STOP statement displays the specified message at your terminal, terminates program execution, and returns control to the operating system.

If you do not specify stop-code, no message is displayed.

Examples

The following examples show valid STOP statements:

```
STOP 98
STOP 'END OF RUN'

DO
  READ *, X, Y
  IF (X > Y) STOP 5555
```

```
END DO
```

Chapter 8. Program Units and Procedures

8.1. Overview

A Fortran 95/90 program consists of one or more program units. There are four types of program units:

- Main program

  The program unit that denotes the beginning of execution. It may or may not have a PROGRAM statement as its first statement.

- External procedures

  Program units that are either user-written functions or subroutines.

- Modules

  Program units that contain declarations, type definitions, procedures, or interfaces that can be shared by other program units.

- Block data program units

  Program units that provide initial values for variables in named common blocks.

A program unit does not have to contain executable statements; for example, it can be a module containing interface blocks for subroutines.

A procedure can be invoked during program execution to perform a specific task. There are several kinds of procedures, as follows:

Kind of Procedure	**Description**
External Procedure	A procedure that is not part of any other program unit.
Module Procedure	A procedure defined within a module
Internal Procedure[1]	A procedure (other than a statement function) contained within a main program, function, or subroutine
Intrinsic Procedure	A procedure defined by the Fortran language
Dummy Procedure	A dummy argument specified as a procedure or appearing in a procedure reference
Statement function	A computing procedure defined by a single statement

[1]The program unit that contains an internal procedure is called its **host**.

A **function** is invoked in an expression using the name of the function or a defined operator. It returns a single value (function result) that is used to evaluate the expression.

A **subroutine** is invoked in a CALL statement or by a defined assignment statement. It does not directly return a value, but values can be passed back to the calling program unit through arguments (or variables) known to the calling program.

Recursion (direct or indirect) is permitted for functions and subroutines.

A procedure interface refers to the properties of a procedure that interact with or are of concern to the calling program. A procedure interface can be explicitly defined in interface blocks. All program units, except block data program units, can contain interface blocks.

For More Information:

- On an overview of program structure, see Section 2.1.
- On intrinsic procedures, see Chapter 9.
- On the scope of program entities, see Section 15.2.
- On recursion, see Section 8.5.1.1.

8.2. Main Program

A Fortran program must include one main program. It takes the following form:

```
[PROGRAM name]
  [specification-part]
  [execution-part]
[CONTAINS
  internal-subprogram-part]
END [PROGRAM [name]]
```

`name`

Is the name of the program.

`specification-part`

Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- PUBLIC and PRIVATE (or their equivalent attributes)

An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.

`execution-part`

Is one or more executable constructs or statements, except for ENTRY or RETURN statements.

`internal-subprogram-part`

Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Rules and Behavior

The PROGRAM statement is optional. Within a program unit, a PROGRAM statement can be preceded only by comment lines or an OPTIONS statement.

The END statement is the only required part of a program. If a name follows the END statement, it must be the same as the name specified in the PROGRAM statement.

The program name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A main program must not reference itself (either directly or indirectly).

Examples

The following is an example of a main program:

```
PROGRAM TEST
  INTEGER C, D, E(20,20)      ! Specification part
  CALL SUB_1                  ! Executable part
...
CONTAINS
  SUBROUTINE SUB_1            ! Internal subprogram
  ...
  END SUBROUTINE SUB_1
END PROGRAM TEST
```

For More Information:

On the default name for a main program, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.3. Modules and Module Procedures

A **module** contains specifications and definitions that can be used by one or more program units. For the module to be accessible, the other program units must reference its name in a USE statement, and the module entities must be public.

A module takes the following form:

```
MODULE name
  [specification-part]
  [CONTAINS
module-subprogram
  [module-subprogram]...]
END [MODULE [name]]
```

`name`

Is the name of the module.

`specification-part`

Is one or more specification statements, except for the following:

- ENTRY
- FORMAT
- AUTOMATIC (or its equivalent attribute)

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- Statement functions

An automatic object must not appear in a specification statement.

`module-subprogram`

Is a function or subroutine subprogram that defines the **module procedure**. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

A module subprogram can contain internal procedures.

Rules and Behavior

If a name follows the END statement, it must be the same as the name specified in the MODULE statement.

The module name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A module is host to any module procedures it contains, and entities in the module are accessible to the module procedures through host association.

A module must not reference itself (either directly or indirectly).

You can use the PRIVATE attribute to restrict access to procedures or variables within a module.

Although ENTRY statements, FORMAT statements, and statement functions are not allowed in the specification part of a module, they are allowed in the specification part of a module subprogram.

Any executable statements in a module can only be specified in a module subprogram.

A module can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

When creating a MODULE that contains datatype declarations, it is recommended that such declarations explicitly specify the kind of the datatype. If an explicit kind is omitted, the MODULE's declarations will be interpreted according to the command-line options in effect when the MODULE is imported, which may result in unintended behavior.

Every module subprogram of any HPF module must be of the same extrinsic kind as its host, and any module subprogram whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

Examples

The following example shows a simple module that can be used to provide global data:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
```

```
SUBROUTINE SUB_Z
  USE MOD_A               ! Makes scalar variables B and C, and array
  ...                     !   E available to this subroutine
END SUBROUTINE SUB_Z
```

The following example shows a module procedure:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
  USE MOD_A               ! Makes scalar variables B and C, and array
  ...                     !   E available to this subroutine
END SUBROUTINE SUB_Z
```

The following example shows a module containing a derived type:

```
MODULE EMPLOYEE_DATA
  TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=40) NAME
  END TYPE EMPLOYEE
END MODULE
```

The following example shows a module containing an interface block:

```
MODULE ARRAY_CALCULATOR
  INTERFACE
    FUNCTION CALC_AVERAGE(D)
      REAL :: CALC_AVERAGE
      REAL, INTENT(IN) :: D(:)
    END FUNCTION
  END INTERFACE
END MODULE ARRAY_CALCULATOR
```

The following example shows a derived-type definition that is public with components that are private:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

This design allows you to change components of a type without affecting other program units that use the module.

If a derived type is needed in more than one program unit, the definition should be placed in a module and accessed by a USE statement whenever it is needed, as follows:

```
MODULE STUDENTS
```

```
  TYPE STUDENT_RECORD
  ...
  END TYPE
CONTAINS
  SUBROUTINE COURSE_GRADE(...)
  TYPE(STUDENT_RECORD) NAME
  ...
  END SUBROUTINE
END MODULE STUDENTS
...

PROGRAM SENIOR_CLASS
  USE STUDENTS
  TYPE(STUDENT_RECORD) ID
  ...
END PROGRAM
```

Program SENIOR_CLASS has access to type STUDENT_RECORD, because it uses module STUDENTS. Module procedure COURSE_GRADE also has access to type STUDENT_RECORD, because the derived-type definition appears in its host.

For More Information:

- On procedure interfaces, see Section 8.9.
- On the PRIVATE and PUBLIC attributes, see Section 5.16.

8.3.1. Module References

A program unit references a module in a USE statement. This module reference lets the program unit access the public definitions, specifications, and procedures in the module.

Entities in a module are public by default, unless the USE statement specifies otherwise or the PRIVATE attribute is specified for the module entities.

A module reference causes use association between the using program unit and the entities in the module.

For More Information:

- On the USE statement, see Section 8.3.2.
- On the PRIVATE and PUBLIC attributes, see Section 5.16.
- On use association, see Section 15.5.1.2.

8.3.2. USE Statement

The USE statement gives a program unit accessibility to public entities in a module. It takes one of the following forms:

```
USE name [, rename-list]
USE name, ONLY : [only-list]
```

name

Is the name of the module.

rename-list

Is one or more items having the following form:

```
local-name => mod-name
```

local-name

Is the name of the entity in the program unit using the module.

mod-name

Is the name of a public entity in the module.

only-list

Is the name of a public entity in the module or a generic identifier (a generic name, defined operator, or defined assignment).

An entity in the *only-list* can also take the form:

```
[local-name =>] mod-name
```

Rules and Behavior

If the USE statement is specified without the ONLY option, the program unit has access to all public entities in the named module.

If the USE statement is specified with the ONLY option, the program unit has access to only those entities following the option.

If more than one USE statement for a given module appears in a scoping unit, the following rules apply:

- If one USE statement does not have the ONLY option, all public entities in the module are accessible, and any *rename-list*s and *only-list*s are interpreted as a single, concatenated *rename-list*.
- If all the USE statements have ONLY options, all the *only-list*s are interpreted as a single, concatenated *only-list*. Only those entities named in one or more of the *only-list*s are accessible.

If two or more generic interfaces that are accessible in a scoping unit have the same name, the same operator, or are both assignments, they are interpreted as a single generic interface. Otherwise, multiple accessible entities can have the same name only if no reference to the name is made in the scoping unit.

The local names of entities made accessible by a USE statement must not be respecified with any attribute other than PUBLIC or PRIVATE. The local names can appear in namelist group lists, but not in a COMMON or EQUIVALENCE statement.

Examples

The following shows examples of the USE statement:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5), D(100)
END MODULE MOD_A
```

```
...
SUBROUTINE SUB_Y
  USE MOD_A, DX => D, EX => E   ! Array D has been renamed DX and array E
  ...                           ! has been renamed EX. Scalar variables B
END SUBROUTINE SUB_Y            ! and C are also available to this subrou-
...                             ! tine (using their module names).
SUBROUTINE SUB_Z
  USE MOD_A, ONLY: B, C         ! Only scalar variables B and C are
  ...                           !   available to this subroutine
END SUBROUTINE SUB_Z
...
```

The following example shows a module containing common blocks:

```
MODULE COLORS
  COMMON /BLOCKA/ C, D(15)
  COMMON /BLOCKB/ E, F
  ...
END MODULE COLORS
...
FUNCTION HUE(A, B)
  USE COLORS
  ...
END FUNCTION HUE
```

The USE statement makes all of the variables in the common blocks in module COLORS available to the function HUE.

To provide data abstraction, a user-defined data type and operations to be performed on values of this type can be packaged together in a module. The following example shows such a module:

```
ODULE CALCULATION
  TYPE ITEM
    REAL :: X, Y
  END TYPE ITEM

  INTERFACE OPERATOR (+)
    MODULE PROCEDURE ITEM_CALC
  END INTERFACE

CONTAINS
  FUNCTION ITEM_CALC (A1, A2)
    TYPE(ITEM) A1, A2, ITEM_CALC
    ...
  END FUNCTION ITEM_CALC
  ...
END MODULE CALCULATION

PROGRAM TOTALS
USE CALCULATION
TYPE(ITEM) X, Y, Z
  ...
  X = Y + Z
  ...
END
```

The USE statement allows program TOTALS access to both the type ITEM and the extended intrinsic operator + to perform calculations.

8.4. Block Data Program Units

A block data program unit provides initial values for nonpointer variables in named common blocks. It takes the following form:

```
BLOCK DATA [name]
   [specification-part]
END [BLOCK DATA [name]]
```

`name`

Is the name of the block data program unit.

`specification-part`

Is one or more of the following statements:

COMMON	INTRINSIC	STATIC
DATA	PARAMETER	TARGET
Derived-type definition	POINTER	Type declaration[2]
DIMENSION	RECORD[1]	USE[3]
EQUIVALENCE	record structure declaration[1]	
IMPLICIT	SAVE	

[2]Can only contain attributes: DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, or TARGET.

[1]For more information on the RECORD statement and record structure declarations, see Section B.12.

[3]Allows access to only named constants.

Rules and Behavior

A block data program unit need not be named, but there can only be one unnamed block data program unit in an executable program.

If a name follows the END statement, it must be the same as the name specified in the BLOCK DATA statement.

An interface block must not appear in a block data program unit and a block data program unit must not contain any executable statements.

If a DATA statement initializes any variable in a named common block, the block data program unit must have a complete set of specification statements establishing the common block. However, all of the variables in the block do not have to be initialized.

A block data program unit can establish and define initial values for more than one common block, but a given common block can appear in only one block data program unit in an executable program.

The name of a block data program unit can appear in the EXTERNAL statement of a different program unit to force a search of object libraries for the block data program unit at link time.

Examples

The following is an example of a block data program unit:

```
BLOCK DATA BLKDAT
```

```
  INTEGER S,X
  LOGICAL T,W
  DOUBLE PRECISION U
  DIMENSION R(3)
  COMMON /AREA1/R,S,U,T /AREA2/W,X,Y
  DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

For More Information:

- On common blocks, see Section 5.4.
- On the DATA statement, see Section 5.5.
- On the EXTERNAL statement, see Section 5.8.

8.5. Functions, Subroutines, and Statement Functions

Functions, subroutines, and statement functions are user-written subprograms that perform computing procedures. The computing procedure can be either a series of arithmetic operations or a series of Fortran statements. A single subprogram can perform a computing procedure in several places in a program, to avoid duplicating a series of operations or statements in each place.

The following table shows the statements that define these subprograms, and how control is transferred to the subprogram:

Subprogram	**Defining Statements**	**Control Transfer Method**
Function	FUNCTION or ENTRY	Function reference[1]
Subroutine	SUBROUTINE or ENTRY	CALL statement[2]
Statement function	Statement function definition	Function reference

[1] A function can also be invoked by a defined operation (see Section 8.9.4).

[2] A subroutine can also be invoked by a defined assignment (see Section 8.9.5).

A **function reference** is used in an expression to invoke a function; it consists of the function name and its **actual arguments**. The function reference returns a value to the calling expression that is used to evaluate the expression.

The following topics are described in this section:

- General rules for function and subroutine subprograms (Section 8.5.1)
- Functions (Section 8.5.2)
- Subroutines (Section 8.5.3)
- Statement functions (Section 8.5.4)

For More Information:

- On the ENTRY statement, see Section 8.11.
- On the CALL statement, see Section 7.3.

8.5.1. General Rules for Function and Subroutine Subprograms

A subprogram can be an external, module, or internal subprogram. The END statement for an internal or module subprogram must be END SUBROUTINE [name] for a subroutine, or END FUNCTION [name] for a function. In an external subprogram, the SUBROUTINE and FUNCTION keywords are optional.

If a subprogram name appears after the END statement, it must be the same as the name specified in the SUBROUTINE or FUNCTION statement.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use the changed values.

A SUBROUTINE or FUNCTION statement can be optionally preceded by an OPTIONS statement.

Dummy arguments (except for dummy pointers or dummy procedures) can be specified with an intent and can be made optional.

The following sections describe recursion, pure procedures, and user-defined elemental procedures.

For More Information:

- On module procedures, see Section 8.3.
- On internal procedures, see Section 8.7.
- On external procedures, see Section 8.6.
- On argument intent, see Section 5.10.
- On optional arguments, see Section 8.8.1.

8.5.1.1. Recursive Procedures

A recursive procedure can reference itself directly or indirectly. Recursion is permitted if the keyword RECURSIVE is specified in a FUNCTION or SUBROUTINE statement, or if RECURSIVE is specified as a compiler option or in an OPTIONS statement.

If a function is directly recursive and array valued, the keywords RECURSIVE and RESULT must both be specified in the FUNCTION statement.

The procedure interface is explicit within the subprogram in the following cases:

- When RECURSIVE is specified for a subroutine
- When RECURSIVE and RESULT are specified for a function

The keyword RECURSIVE must be specified if any of the following applies (directly or indirectly):

- The subprogram invokes itself.
- The subprogram invokes a subprogram defined by an ENTRY statement in the same subprogram.
- An ENTRY procedure in the same subprogram invokes one of the following:

- Itself
- Another ENTRY procedure in the same subprogram
- The subprogram defined by the FUNCTION or SUBROUTINE statement

For More Information:

- On the FUNCTION statement, see Section 8.5.2.
- On the SUBROUTINE statement, see Section 8.5.3.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the OPTIONS statement, see Section 13.3.

8.5.1.2. Pure Procedures

A pure procedure is a user-defined procedure that is specified by using the prefix PURE (or ELEMENTAL) in a FUNCTION or SUBROUTINE statement. Pure procedures are a feature of Fortran 95.

A pure procedure has no side effects. It has no effect on the state of the program, except for the following:

- For functions: It returns a value.
- For subroutines: It modifies INTENT(OUT) and INTENT(INOUT) parameters.

The following intrinsic and library procedures are implicitly pure:

- All intrinsic functions
- The elemental intrinsic subroutine MVBITS
- The library routines in the HPF_LIBRARY

A statement function is pure only if all functions that it references are pure.

Rules and Behavior

Except for procedure arguments and pointer arguments, the following intent must be specified for all dummy arguments in the specification part of the procedure:

- For functions: INTENT(IN)
- For subroutines: any INTENT (IN, OUT, or INOUT)

A local variable declared in a pure procedure (including variables declared in any internal procedure) must not:

- Specify the SAVE attribute
- Be initialized in a type declaration statement or a DATA statement

The following variables have restricted use in pure procedures (and any internal procedures):

- Global variables
- Dummy arguments with INTENT(IN) (or no declared intent)
- Objects that are storage associated with any part of a global variable

They must not be used in any context that does either of the following:

- Causes their value to change. For example, they must not be used as:
  - The left side of an assignment statement or pointer assignment statement
  - An actual argument associated with a dummy argument with INTENT(OUT), INTENT(INOUT), or the POINTER attribute
  - An index variable in a DO or FORALL statement, or an implied-do clause
  - The variable in an ASSIGN statement
  - An input item in a READ statement
  - An internal file unit in a WRITE statement
  - An object in an ALLOCATE, DEALLOCATE, or NULLIFY statement
  - An IOSTAT or SIZE specifier in an I/O statement, or the STAT specifier in a ALLOCATE or DEALLOCATE statement
- Creates a pointer to that variable. For example, they must not be used as:
  - The target in a pointer assignment statement
  - The right side of an assignment to a derived-type variable (including a pointer to a derived type) if the derived type has a pointer component at any level

A pure procedure must not contain the following:

- Any external I/O statement (including a READ or WRITE statement whose I/O unit is an external file unit number or *)
- A PAUSE statement
- A STOP statement

A pure procedure can be used in contexts where other procedures are restricted; for example:

- It can be called directly in a FORALL statement or be used in the mask expression of a FORALL statement.
- It can be called from a pure procedure. Pure procedures can only call other pure procedures.
- It can be passed as an actual argument to a pure procedure.

If a procedure is used in any of these contexts, its interface must be explicit and it must be declared pure in that interface.

Examples

The following shows a pure function:

```
PURE INTEGER FUNCTION MANDELBROT(X)
  COMPLEX, INTENT(IN) :: X
  COMPLEX  :: XTMP
  INTEGER  :: K
  ! Assume SHARED_DEFS includes the declaration
  ! INTEGER ITOL
  USE SHARED_DEFS

  K = 0
  XTMP = -X
  DO WHILE (ABS(XTMP)<2.0 .AND. K<ITOL)
    XTMP = XTMP**2 - X
    K = K + 1
  END DO
  ITER = K
END FUNCTION
```

The following shows the preceding function used in an interface block:

```
INTERFACE
  PURE INTEGER FUNCTION MANDELBROT(X)
    COMPLEX, INTENT(IN) :: X
  END FUNCTION MANDELBROT
END INTERFACE
```

The following shows a FORALL construct calling the MANDELBROT function to update all the elements of an array:

```
FORALL (I = 1:N, J = 1:M)
  A(I,J) = MANDELBROT(COMPLX((I-1)*1.0/(N-1), (J-1)*1.0/(M-1)))
END FORALL
```

For More Information:

- On the FUNCTION statement, see Section 8.5.2.
- On the SUBROUTINE statement, see Section 8.5.3.
- On pure procedures in FORALLs, see Section 4.2.5.
- On pure procedures in interface blocks, see Section 8.9.2.
- On how to use pure procedures, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.5.1.3. Elemental Procedures

An elemental procedure is a user-defined procedure that is a restricted form of pure procedure. An elemental procedure can be passed an array, which is acted upon one element at a time. Elemental procedures are a feature of Fortran 95.

To specify an elemental procedure, use the prefix ELEMENTAL in a FUNCTION or SUBROUTINE statement.

An explicit interface must be visible to the caller of an ELEMENTAL procedure.

For functions, the result must be scalar; it cannot have the POINTER attribute.

Dummy arguments have the following restrictions:

- They must be scalar.
- They cannot have the POINTER attribute.
- They (or their subobjects) cannot appear in a specification expression, except as an argument to one of the intrinsic functions BIT_SIZE, LEN, KIND, or the numeric inquiry functions.
- They cannot be *.
- They cannot be dummy procedures.

If the actual arguments are all scalar, the result is scalar. If the actual arguments are array-valued, the values of the elements (if any) of the result are the same as if the function or subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

Elemental procedures are pure procedures and all rules that apply to pure procedures also apply to elemental procedures.

Examples

Consider the following:

```
MIN (A, 0, B)              ! A and B are arrays of shape (S, T)
```

In this case, the elemental reference to the MIN intrinsic function is an array expression whose elements have the following values:

```
MIN (A(I,J), 0, B(I,J)), I = 1, 2, ..., S, J = 1, 2, ..., T
```

For More Information:

- On the FUNCTION statement, see Section 8.5.2.
- On the SUBROUTINE statement, see Section 8.5.3.
- On determining when procedures require explicit interfaces, see Section 8.9.1.
- On pure procedures and the prefix PURE, see Section 8.5.1.2.
- On optional arguments, see Section 8.8.1.

8.5.2. Functions

A **function** subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression.

The FUNCTION statement is the initial statement of a function subprogram. It takes the following form:

```
[prefix] FUNCTION name ([d-arg-list]) [RESULT (r-name)]
```

`prefix`

Is one of the following:

```
type [keyword]
keyword [type]
```

`type`

Is a data type specifier.

`keyword`

Is one of the following:

Keyword	Meaning
RECURSIVE	Permits direct recursion to occur. If a function is directly recursive and array valued, RESULT must also be specified (see Section 8.5.1.1).
PURE	Asserts that the procedure has no side effects (see Section 8.5.1.2).
ELEMENTAL	Restricted form of pure procedure that acts on one array element at a time (see Section 8.5.1.3).

`name`

Is the name of the function. If RESULT is specified, the function name must not appear in any specification statement in the scoping unit of the function subprogram.

The function name can be followed by the length of the data type. The length is specified by an asterisk (*) followed by any unsigned, nonzero integer that is a valid length for the function's type. For example, REAL FUNCTION LGFUNC*8 (Y, Z) specifies the function result as REAL(8) (or REAL*8).

This optional length specification is not permitted if the length has already been specified following the keyword CHARACTER.

`d-arg-list`

Is a list of one or more dummy arguments.

`r-name`

Is the name of the function result. This name must not be the same as the function name.

Rules and Behavior

The type and kind parameters (if any) of the function's result can be defined in the FUNCTION statement or in a type declaration statement within the function subprogram, but not both. If no type is specified, the type is determined by implicit typing rules in effect for the function subprogram.

Execution begins with the first executable construct or statement following the FUNCTION statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

If you specify CHARACTER* (*), the function assumes the length declared for it in the program unit that invokes it. This type of character function can have different lengths when it is invoked by different program units; it is an obsolescent feature in Fortran 95.

If the length is specified as an integer constant, the value must agree with the length of the function specified in the program unit that invokes the function. If no length is specified, a length of 1 is assumed.

If the function is array-valued or a pointer, the declarations within the function must state these attributes for the function result name. The specification of the function result attributes, dummy argument attributes, and the information in the procedure heading collectively define the interface of the function.

The value of the result variable is returned by the function when it completes execution. Certain rules apply depending on whether the result is a pointer, as follows:

- If the result is a pointer, its allocation status must be determined before the function completes execution. (The function must associate a target with the pointer, or cause the pointer to be explicitly disassociated from a target.)

  The shape of the value returned by the function is determined by the shape of the result variable when the function completes execution.

- If the result is not a pointer, its value must be defined before the function completes execution. If the result is an array, all the elements must be defined; if the result is a derived-type structure, all the components must be defined.

A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA statement, a PROGRAM statement, or another FUNCTION statement. ENTRY statements can be included to provide multiple entry points to the subprogram.

You can use a CALL statement to invoke a function as long as write the function is not one of the following types:

- REAL(8)
- REAL(16)
- COMPLEX(8)
- COMPLEX(16)
- CHARACTER

Section 8.5.2.1 describes the RESULT keyword and Section 8.5.2.2 describes function references.

Examples

The following example uses the Newton-Raphson iteration method (F(X) = cosh(X) + cos(X) - A = 0) to get the root of the function:

```
FUNCTION ROOT(A)
  X  = 1.0
  DO
    EX = EXP(X)
    EMINX = 1./EX
    ROOT  = X - ((EX+EMINX)*.5+COS(X)-A)/((EX-EMINX)*.5-SIN(X))
    IF (ABS((X-ROOT)/ROOT) .LT. 1E-6) RETURN
    X  = ROOT
  END DO
END
```

In the preceding example, the following formula is calculated repeatedly until the difference between X_i and X_{i+1} is less than 1.0E–6:

$$X_{i+1} = X_i - \frac{\cosh(X_i) + \cos(X_i) - A}{\sinh(X_i) - \sin(X_i)}$$

The following example shows an assumed-length character function:

```
CHARACTER*(*) FUNCTION REDO(CARG)
  CHARACTER*1 CARG
  DO I=1,LEN(REDO)
    REDO(I:I) = CARG
  END DO
  RETURN
END FUNCTION
```

This function returns the value of its argument, repeated to fill the length of the function.

Within any given program unit, all references to an assumed-length character function must have the same length. In the following example, the REDO function has a length of 1000:

```
CHARACTER*1000 REDO, MANYAS, MANYZS
MANYAS = REDO('A')
MANYZS = REDO('Z')
```

Another program unit within the executable program can specify a different length. For example, the following REDO function has a length of 2:

```
CHARACTER HOLD*6, REDO*2
HOLD = REDO('A')//REDO('B')//REDO('C')
```

The following example shows a dynamic array-valued function:

```
FUNCTION SUB (N)
  REAL, DIMENSION(N) :: SUB
  ...
END FUNCTION
```

For More Information:

- On general rules that apply to function subprograms, see Section 8.5.1.
- On argument keywords in function references, see Section 8.5.2.2.
- On the ENTRY statement, see Section 8.11.
- On the RETURN statement, see Section 7.10.
- On obsolescent features in Fortran 95, see Appendix A.

8.5.2.1. RESULT Keyword

Normally, a function result is returned in the function's name, and all references to the function name are references to the function result.

However, if you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. In this case, all references to the function name are recursive calls, and the function name must not appear in specification statements.

The RESULT name must be different from the name of the function.

The following shows an example of a recursive function specifying a RESULT variable:

```
RECURSIVE FUNCTION FACTORIAL(P) RESULT(L)
  INTEGER, INTENT(IN) :: P
  INTEGER L
  IF (P == 1) THEN
    L = 1
  ELSE
    L = P * FACTORIAL(P - 1)
  END IF
END FUNCTION
```

8.5.2.2. Function References

Functions are invoked by a function reference in an expression or by a defined operation.

A function reference takes the following form:

```
fun ([a-arg [,a-arg]...])
```

`fun`

Is the name of the function subprogram.

`a-arg`

Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the function. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure).

Rules and Behavior

When a function is referenced, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

Execution of the function produces a result that is assigned to the function name or to the result name, depending on whether the RESULT keyword was specified.

The program unit uses the result value to complete the evaluation of the expression containing the function reference.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

If a dummy argument is specified with the INTENT attribute, its use may be limited. A dummy argument whose intent is not specified is subject to the limitations of its associated actual argument.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Table 9.1).

Examples

Consider the following example:

```
X = 2.0
NEW_COS = COS(X)         ! A function reference
```

Intrinsic function COS calculates the cosine of 2.0. The value –0.4161468 is returned (in place of COS(X)) and assigned to NEW_COS.

For More Information:

- On the INTENT attribute, see Section 5.10.
- On defined operations, see Section 8.9.4.
- On procedure arguments, see Section 8.8.
- On dummy arguments, see Section 8.8.7.

- On intrinsic functions, see Chapter 9.
- On optional arguments, see Section 8.8.1.
- On the RESULT keyword in FUNCTION statements, see Section 8.5.2.1.
- On the FUNCTION statement, see Section 8.5.2.

8.5.3. Subroutines

A **subroutine** subprogram is invoked in a CALL statement or by a defined assignment statement, and does not return a particular value.

The SUBROUTINE statement is the initial statement of a subroutine subprogram. It takes the following form:

```
[prefix] SUBROUTINE name [([d-arg-list])]
```

`prefix`

Is one of the following:

Keyword	Meaning
RECURSIVE	Permits direct recursion to occur (see Section 8.5.1.1).
PURE	Asserts that the procedure has no side effects (see Section 8.5.1.2).
ELEMENTAL	Restricted form of pure procedure that acts on one array element at a time (see Section 8.5.1.3).

`name`

Is the name of the subroutine.

`d-arg-list`

Is a list of one or more dummy arguments or alternate return specifiers (*).

Rules and Behavior

A subroutine is invoked by a CALL statement or defined assignment. When a subroutine is invoked, dummy arguments (if present) become associated with the corresponding actual arguments specified in the call.

Execution begins with the first executable construct or statement following the SUBROUTINE statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

A subroutine subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, a PROGRAM statement, or another SUBROUTINE statement. ENTRY statements can be included to provide multiple entry points to the subprogram.

Examples

The following example shows a subroutine:

Main Program	Subroutine
CALL HELLO_WORLD ... END	SUBROUTINE HELLO_WORLD PRINT *, "Hello World" END SUBROUTINE

The following example uses alternate return specifiers to determine where control transfers on completion of the subroutine:

Main Program	Subroutine
CALL CHECK(A,B,*10,*20,C) TYPE *, 'VALUE LESS THAN ZERO' GO TO 30 10 TYPE*, 'VALUE EQUALS ZERO' GO TO 30 20 TYPE*, 'VALUE MORE THAN ZERO' 30 CONTINUE ...	SUBROUTINE CHECK(X,Y,*,*,Q) ... 50 IF (Z) 60,70,80 60 RETURN 70 RETURN 1 80 RETURN 2 END

The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list.

The value of Z determines the return, as follows:

- If Z < zero, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the main program.
- If Z == zero, the return is to statement label 10 in the main program.
- If Z > zero, the return is to statement label 20 in the main program.

(An alternate return is an obsolescent feature in Fortran 95 and Fortran 90).

For More Information:

- On general rules that apply to subroutine subprograms, see Section 8.5.1.
- On the CALL statement, see Section 7.3.
- On argument keywords in subroutine references, see Section 7.3.
- On defined assignment, see Section 8.9.5.
- On the RETURN statement, see Section 7.10.
- On procedure arguments, see Section 8.8.
- On intrinsic subroutines, see Chapter 9.
- On the ENTRY statement, see Section 8.11.
- On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

8.5.4. Statement Functions

A statement function is a procedure defined by a single statement in the same program unit in which the procedure is referenced. It takes the following form:

```
fun ([d-arg [,d-arg]...]) = expr
```

fun

Is the name of the statement function.

d-arg

Is a dummy argument. A dummy argument can appear only once in any list of dummy arguments, and its scope is local to the statement function.

expr

Is a scalar expression defining the computation to be performed.

Named constants and variables used in the expression must have been declared previously in the specification part of the scoping unit or made accessible by use or host association.

If the expression contains a function reference, the function must have been defined previously in the same program unit.

A statement function reference takes the following form:

```
fun ([a-arg [,a-arg]...])
```

fun

Is the name of the statement function.

a-arg

Is an actual argument.

Rules and Behavior

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of a statement function can be explicitly defined in a type declaration statement. If no type is specified, the type is determined by implicit typing rules in effect for the program unit.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

Except for the data type, declarative information associated with an entity is not associated with dummy arguments in the statement function; for example, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.

The name of the statement function cannot be the same as the name of any other entity within the same program unit.

Any reference to a statement function must appear in the same program unit as the definition of that function.

A statement function reference must appear as (or be part of) an expression. The reference cannot appear on the left side of an assignment statement.

A statement function must not be provided as a procedure argument.

Examples

The following are examples of statement functions:

```
REAL VOLUME, RADIUS
VOLUME(RADIUS) = 4.189*RADIUS**3

CHARACTER*10 CSF,A,B
CSF(A,B) = A(6:10)//B(1:5)
```

The following example shows a statement function and some references to it:

```
AVG(A,B,C) = (A+B+C)/3.
...
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q) .LT. AVG(X,Y,Z)) STOP
FINAL = AVG(TEST3,TEST4,LAB2)       ! Invalid reference; implicit
...                                 ! type of third argument does not
...                                 ! match implicit type of dummy argument
```

Implicit typing problems can be avoided if all arguments are explicitly typed.

The following statement function definition is invalid because it contains a constant, which cannot be used as a dummy argument:

```
REAL COMP, C, D, E
COMP(C,D,E,3.) = (C + D - E)/3.
```

For More Information:

- On procedure arguments, see Section 8.8.
- On use and host association, see Section 15.5.1.2.

8.6. External Procedures

External procedures are user-written functions or subroutines. They are located outside of the main program and can't be part of any other program unit.

External procedures can be invoked by the main program or any procedure of an executable program.

In Fortran 95/90, external procedures can include internal procedures, as long as the internal procedures appear between a CONTAINS statement and the end of the procedure.

An external procedure can reference itself (directly or indirectly).

The interface of an external procedure is implicit unless an interface block is supplied for the procedure.

For More Information:

- On function and subroutine subprograms, see Section 8.5.
- On procedure interfaces, see Section 8.9.

- On passing arguments, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.7. Internal Procedures

Internal procedures are functions or subroutines that follow a CONTAINS statement in a program unit. The program unit in which the internal procedure appears is called its host.

Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram.

An internal procedure takes the following form:

```
CONTAINS
   internal-subprogram
   [internal-subprogram]...
```

`internal-subprogram`

Is a function or subroutine subprogram that defines the procedure. An internal subprogram must not contain any other internal subprograms.

Rules and Behavior

Internal procedures are the same as external procedures, except for the following:

- Only the host program unit can use an internal procedure.
- An internal procedure has access to host entities by host association; that is, names declared in the host program unit are useable within the internal procedure.
- In Fortran 95/90, the name of an internal procedure must not be passed as an argument to another procedure. However, VSI Fortran allows an internal procedure name to be passed as an actual argument to another procedure.
- An internal procedure must not contain an ENTRY statement.

An internal procedure can reference itself (directly or indirectly); it can be referenced in the execution part of its host and in the execution part of any internal procedure contained in the same host (including itself).

The interface of an internal procedure is always explicit.

Every HPF internal subprogram must be of the same extrinsic kind as its host, and any internal subprogram whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

Examples

The following example shows an internal procedure:

```
PROGRAM COLOR_GUIDE
...
CONTAINS
  FUNCTION HUE(BLUE)   ! An internal procedure
  ...
  END FUNCTION HUE
```

```
END PROGRAM
```

For More Information:

- On function and subroutine subprograms, see Section 8.5.
- On host association, see Section 15.5.1.2.
- On procedure interfaces, see Section 8.9.

8.8. Argument Association

Procedure arguments provide a way for different program units to access the same data.

When a procedure is referenced in an executable program, the program unit invoking the procedure can use one or more actual arguments to pass values to the procedure's dummy arguments. The dummy arguments are associated with their corresponding actual arguments when control passes to the subprogram.

In general, when control is returned to the calling program unit, the last value assigned to a dummy argument is assigned to the corresponding actual argument.

An actual argument can be a variable, expression, or procedure name. The type and kind parameters, and rank of the actual argument must match those of its associated dummy argument.

A dummy argument is either a dummy data object, a dummy procedure, or an alternate return specifier (*). Except for alternate return specifiers, dummy arguments can be optional.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments.

A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

A scalar dummy argument can be associated with only a scalar actual argument.

If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays to process arrays of different sizes in a single subprogram.

A dummy argument referenced as a subprogram must be associated with an actual argument that has been declared EXTERNAL or INTRINSIC in the calling routine.

If a scalar dummy argument is of type character, its length must not be greater than the length of its associated actual argument.

If the character dummy argument's length is specified as * (*) (assumed length), it uses the length of the associated actual argument.

Once an actual argument has been associated with a dummy argument, no action can be taken that affects the value or availability of the actual argument, except indirectly through the dummy argument. For example, if the following statement is specified:

```
CALL SUB_A (B(2:6), B(4:10))
```

B(4:6) must not be defined, redefined, or become undefined through either dummy argument, since it is associated with both arguments. However, B(2:3) is definable through the first argument, and B(7:10) is definable through the second argument.

Similarly, if any part of the actual argument is defined through a dummy argument, the actual argument can only be referenced through that dummy argument during execution of the procedure. For example, if the following statements are specified:

```
MODULE MOD_A
  REAL :: A, B, C, D
END MODULE MOD_A

PROGRAM TEST
  USE MOD_A
  CALL SUB_1 (B)
  ...
END PROGRAM TEST

SUBROUTINE SUB_1 (F)
  USE MOD_A
  ...
  WRITE (*,*) F
END SUBROUTINE SUB_1
```

Variable B must not be directly referenced during the execution of SUB_1 because it is being defined through dummy argument F. However, B can be indirectly referenced through F (and directly referenced when SUB_1 completes execution).

The following sections provide more details on arguments:

- Optional arguments (Section 8.8.1)
- The different kinds of arguments
  - Array arguments (Section 8.8.2)
  - Pointer arguments (Section 8.8.3)
  - Assumed-length character arguments (Section 8.8.4)
  - Character constant and Hollerith arguments (Section 8.8.5)
  - Alternate return arguments (Section 8.8.6)
  - Dummy procedure arguments (Section 8.8.7)
- References to generic procedures (Section 8.8.8)
- References to non-Fortran procedures (Section 8.8.9)

For More Information:

- On argument keywords in subroutine references, see Section 7.3.
- On argument keywords in function references, see Section 8.5.2.2.
- On built-in functions to pass actual arguments, see Section 8.8.9.1.

8.8.1. Optional Arguments

Dummy arguments can be made optional if they are declared with the OPTIONAL attribute. In this case, an actual argument does not have to be supplied for it in a procedure reference.

Positional arguments (if any) must appear first in an actual argument list, followed by keyword arguments (if any). If an optional argument is the last positional argument, it can simply be omitted if desired.

However, if the optional argument is to be omitted but it is not the last positional argument, keyword arguments must be used for any subsequent arguments in the list.

The following example shows optional arguments:

```
PROGRAM RESULT
TEST_RESULT = LGFUNC(A, B=D)
...
CONTAINS
  FUNCTION LGFUNC(G, H, B)
    OPTIONAL H, B
    ...
  END FUNCTION
END
```

In the function reference, A is a positional argument associated with required dummy argument G. The second actual argument D is associated with optional dummy argument B by its keyword name (B). No actual argument is associated with optional argument H.

There are two intrinsics you can use to determine arguments:

- PRESENT (see Section 8.8.1.1)
- IARGCOUNT (see Section 8.8.1.2)

8.8.1.1. Using the PRESENT Intrinsic Function

You can use the PRESENT intrinsic function to determine if an actual argument is associated with an optional dummy argument in a particular reference.

Optional arguments must be defined in explicit procedure interfaces so that appropriate argument associations can be made for the PRESENT to work.

See Example 8.1.

Example 8.1. Use of the PRESENT Intrinsic With a Defined Interface

```
!  Compile /NOOPT to avoid inlining
!
           SUBROUTINE CHECK (X, Y)
             REAL X, Z
             REAL, OPTIONAL :: Y

             IF (PRESENT (Y)) THEN
               WRITE(6,10)
10             FORMAT(1X, "Y is present")
               Z = Y
```

```
              ELSE
                WRITE(6,20)
20              FORMAT(1X, "Y is NOT present")
                Z = X * 2
              END IF
              TYPE *,Z
            END

            PROGRAM MAIN
!
! Define CHECK's interface here inside the caller, so MAIN knows how to
 call it
!
            INTERFACE
                 SUBROUTINE CHECK(U,V)
                 REAL U
                 REAL, OPTIONAL :: V
                 END SUBROUTINE
            END INTERFACE

            WRITE (6,100)
100         FORMAT(1X, "Call with a Y")
            CALL CHECK (15.0, 12.0)      ! Causes B to be set to 12.0
            WRITE (6,200)
200         FORMAT(1X, "Call without a Y")
            CALL CHECK (15.0)            ! Causes B to be set to 30.0
            END
$ f90/noop example
$ lin example
$ r example
Call with a Y
Y is present
  12.00000
Call without a Y
Y is NOT present
  30.00000
$
```

The implementation of PRESENT depends on the caller passing a null reference value for any omitted actual argument. This is true even for trailing omitted actual arguments. In this regard, the PRESENT intrinsic does not take advantage of the shortened argument list convention allowed in the OpenVMS Calling Standard. On the calling side, it is the explicit declaration of the full interface that tells the caller how many actual arguments must be provided in any call, even when fewer arguments are written in the source.

8.8.1.2. Using the IARGCOUNT Intrinsic Function

You can use the IARGCOUNT intrinsic function to return the count of actual arguments passed to the routine. With IARGCOUNT, there is no requirement for the caller to see an explicit interface.

See Example 8.2.

Example 8.2. Use of the IARGCOUNT Intrinsic

```
!  Compile /NOOPT to prevent inlining !
!
            SUBROUTINE CHECK (X, Y)
```

```
              REAL X, Z
              REAL, OPTIONAL :: Y

              IF (IARGCOUNT() .GT. 1) THEN
                WRITE(6,10)
10              FORMAT(1X, "Y is present")
                Z = Y
              ELSE
                WRITE(6,20)
20              FORMAT(1X, "Y is NOT present")
                Z = X * 2
              END IF
              TYPE *,Z
            END

            PROGRAM MAIN
            INTEGER I
            CHARACTER C(4)
            REAL    R
            EQUIVALENCE(I,C,R)

            WRITE (6,100)
100         FORMAT(1X, "Call with a Y")
            CALL CHECK (15.0, 12.0)      ! Causes B to be set to 12.0
            WRITE (6,200)
200         FORMAT(1X,"Call without a Y")
            CALL CHECK (15.0)            ! Causes B to be set to 30.0
            END
$ f90/noop example2
$ lin example2
$ r example2
Call with a Y
Y is present
  12.00000
Call without a Y
Y is NOT present
  30.00000
$
```

For More Information:

- On general rules for procedure argument association, see Section 8.8.
- On the OPTIONAL attribute, see Section 5.13.
- On argument keywords in subroutine references, see Section 7.3.
- On argument keywords in function references, see Section 8.5.2.2.
- On the PRESENT intrinsic function, see Section 9.4.117.
- On the IARGCOUNT intrinsic function, see Section 9.4.58.

8.8.2. Array Arguments

Arrays are sequences of elements. Each element of an actual array is associated with the element of the dummy array that has the same position in array element order.

If the dummy argument is an explicit-shape or assumed-size array, the size of the dummy argument array must not exceed the size of the actual argument array.

The type and kind parameters of an explicit-shape or assumed-size dummy argument must match the type and kind parameters of the actual argument, but their ranks need not match.

If the dummy argument is an assumed-shape array, the size of the dummy argument array is equal to the size of the actual argument array. The associated 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 actual argument is an array section with a vector subscript, the associated dummy argument must not be defined.

The declaration of an array used as a dummy argument can specify the lower bound of the array.

Although most types of arrays can be used as dummy arguments, allocatable arrays cannot be dummy arguments. Allocatable arrays can be used as actual arguments.

Dummy argument arrays declared as assumed-shape, deferred-shape, or pointer arrays require an explicit interface visible to the caller.

For More Information:

- On general rules for procedure argument association, see Section 8.8.
- On arrays, see Section 3.5.2.
- On assumed-shape arrays, see Section 5.1.4.2.
- On array element order, see Section 3.5.2.2.
- On array association, see Section 15.5.3.2.
- On explicit-shape arrays, see Section 5.1.4.1.
- On assumed-size arrays, see Section 5.1.4.3.

8.8.3. Pointer Arguments

An argument is a pointer if it is declared with the POINTER attribute.

When a procedure is invoked, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target.

If both the dummy and actual arguments are pointers, an explicit interface is required.

A dummy argument that is a pointer can be associated only with an actual argument that is a pointer. However, an actual argument that is a pointer can be associated with a nonpointer dummy argument. In this case, the actual argument is associated with a target and the dummy argument, through argument association, also becomes associated with that target.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument when the procedure is invoked.

If the dummy argument has the TARGET attribute, and is either a scalar or assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, the following occurs:

- Any pointer associated with the actual argument becomes associated with the corresponding dummy argument when the procedure is invoked.
- Any pointers associated with the dummy argument remain associated with the actual argument when execution of the procedure completes.

If the dummy argument has the TARGET attribute, and is an explicit-shape or assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, association of actual and corresponding dummy arguments when the procedure is invoked or when execution is completed is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have that attribute or is an array section with a vector subscript, any pointer associated with the dummy argument becomes undefined when execution of the procedure completes.

For More Information:

- On general rules for procedure argument association, see Section 8.8.
- On pointers, see Section 5.15.
- On pointer assignment, see Section 4.2.3.
- On the TARGET attribute, see Section 5.18.
- On passing pointers as arguments, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.8.4. Assumed-Length Character Arguments

An assumed-length character argument is a dummy argument that assumes the length attribute of its corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

A character array dummy argument can also have an assumed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The assumed length and the array declarator together determine the size of the assumed-length character array.

The following example shows an assumed-length character argument:

```
INTEGER FUNCTION ICMAX(CVAR)
  CHARACTER*(*) CVAR
  ICMAX = 1
  DO I=2,LEN(CVAR)
    IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
  END DO
  RETURN
END
```

The function ICMAX finds the position of the character with the highest ASCII code value. It uses the length of the assumed-length character argument to control the iteration. Intrinsic function LEN determines the length of the argument.

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
...
I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX(VAR(3:4)//CARRAY(3,5))
```

For More Information:

- On the LEN intrinsic function, see Section 9.4.81.
- On general rules for procedure argument association, see Section 8.8.

8.8.5. Character Constant and Hollerith Arguments

If an actual argument is a character constant (for example, 'ABCD'), the corresponding dummy argument must be of type character. If an actual argument is a Hollerith constant (for example, 4HABCD), the corresponding dummy argument must have a numeric data type.

The following example shows character and Hollerith constants being used as actual arguments:

```
SUBROUTINE S(CHARSUB, HOLLSUB, A, B)
EXTERNAL CHARSUB, HOLLSUB
...
CALL CHARSUB(A, 'STRING')
CALL HOLLSUB(B, 6HSTRING)
```

The subroutines CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument 'STRING' in the call to CHARSUB must correspond to a character dummy argument, and the actual argument 6HSTRING in the call to HOLLSUB must correspond to a numeric dummy argument.

For More Information:

On general rules for procedure argument association, see Section 8.8.

8.8.6. Alternate Return Arguments

Alternate return (dummy) arguments can appear in a subroutine argument list. They cause execution to transfer to a labeled statement rather than to the statement immediately following the statement that called the routine. The alternate return is indicated by an asterisk (*). (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90).

There can be any number of alternate returns in a subroutine argument list, and they can be in any position in the list.

An actual argument associated with an alternate return dummy argument is called an alternate return specifier; it is indicated by an asterisk (*), or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement.

Alternate returns cannot be declared optional.

In Fortran 95/90, you can also use the RETURN statement to specify alternate returns.

The following example shows alternate return actual and dummy arguments:

```
CALL MINN(X, Y, *300, *250, Z)
....
SUBROUTINE MINN(A, B, *, *, C)
```

For More Information:

- On general rules for procedure argument association, see Section 8.8.
- On subroutine subprograms, see Section 8.5.3.
- On the CALL statement, see Section 7.3.
- On the RETURN statement, see Section 7.10.
- On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

8.8.7. Dummy Procedure Arguments

If an actual argument is a procedure, its corresponding dummy argument is a dummy procedure. Dummy procedures can appear in function or subroutine subprograms.

The actual argument must be the specific name of an external, module, intrinsic, or another dummy procedure. If the specific name is also a generic name, only the specific name is associated with the dummy argument. Not all specific intrinsic procedures can appear as actual arguments. (For more information, see Table 9.1).

The actual argument and corresponding dummy procedure must both be subroutines or both be functions.

If the interface of the dummy procedure is explicit, the type and kind parameters, and rank of the associated actual procedure must be the same as that of the dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a subroutine, the actual argument must be a subroutine or a dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a function or is explicitly typed, the actual argument must be a function or a dummy procedure.

Dummy procedures can be declared optional, but they must not be declared with an intent.

The following is an example of a procedure used as an argument:

```
REAL FUNCTION LGFUNC(BAR)
  INTERFACE
    REAL FUNCTION BAR(Y)
      REAL, INTENT(IN) :: Y
    END
  END INTERFACE
  ...
  LGFUNC = BAR(2.0)
  ...
END FUNCTION LGFUNC
```

For More Information:

On general rules for procedure argument association, see Section 8.8.

8.8.8. References to Generic Procedures

Generic procedures are procedures with different specific names that can be accessed under one generic (common) name. In FORTRAN 77, generic procedures were limited to intrinsic procedures. In Fortran 95/90, you can use generic interface blocks to specify generic properties for intrinsic and user-defined procedures.

If you refer to a procedure by using its generic name, the selection of the specific routine is based on the number of arguments and the type and kind parameters, and rank of each argument.

All procedures given the same generic name must be subroutines, or all must be functions. Any two must differ enough so that any invocation of the procedure is unambiguous.

The following sections describe references to generic intrinsic functions and show an example of using intrinsic function names.

For More Information:

- On user-defined generic procedures, see Section 8.9.3.
- On the rules for unambiguous procedure references, see Section 15.3.
- On the rules for resolving ambiguous procedure references, see Section 15.4.
- On intrinsic procedures, see Chapter 9.

8.8.8.1. References to Generic Intrinsic Functions

The generic intrinsic function name COS lists six specific intrinsic functions that calculate cosines: COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS. These functions return different values: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), and COMPLEX(16), respectively.

If you invoke the cosine function by using the generic name COS, the compiler selects the appropriate routine based on the arguments that you specify. For example, if the argument is REAL(4), COS is selected; if it is REAL(8), DCOS is selected; and if it is COMPLEX(4), CCOS is selected.

You can also explicitly refer to a particular routine. For example, you can invoke the double-precision cosine function by specifying DCOS.

Procedure selection occurs independently for each generic reference, so you can use a generic reference repeatedly in the same program unit to access different intrinsic procedures.

You cannot use generic function names to select intrinsic procedures if you use them as follows:

- The name of a statement function
- A dummy argument name, a common block name, or a variable or array name

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Not all specific intrinsic functions can appear as actual arguments. (For more information, see Table 9.1).

Generic procedure names are local to the program unit that refers to them, so they can be used for other purposes in other program units.

Normally, an intrinsic procedure name refers to the Fortran 95/90 library procedure with that name. However, the name can refer to a user-defined procedure when the name appears in an EXTERNAL statement.

Note

If you call an intrinsic procedure by using the wrong number of arguments or an incorrect argument type, the compiler assumes you are referring to an external procedure. For example, intrinsic procedure SIN requires one argument; if you specify two arguments, such as SIN(10,4), the compiler assumes SIN is external and not intrinsic.

Except when used in an EXTERNAL statement, intrinsic procedure names are local to the program unit that refers to them, so they can be used for other purposes in other program units. The data type of an intrinsic procedure does not change if you use an IMPLICIT statement to change the implied data type rules.

Intrinsic and user-defined procedures cannot have the same name if they appear in the same program unit.

Examples

Example 8.3 shows the local and global properties of an intrinsic function name. It uses intrinsic function SIN as the:

- Name of a statement function
- Generic name of an intrinsic function
- Specific name of an intrinsic function
- Name of a user-defined function

Example 8.3. Using and Redefining an Intrinsic Function Name

```
!     Compare ways of computing sine

      PROGRAM SINES
        DOUBLE PRECISION X, PI
        PARAMETER (PI=3.14159265358979323 8D0)
        COMMON V(3)

❶ !     Define SIN as a statement function

        SIN(X) = COS(PI/2-X)
        DO X = -PI, PI, 2*PI/100

❷ !     Reference the statement function SIN

          WRITE (6,100) X, V, SIN(X)
        END DO
        CALL COMPUT(X)
100     FORMAT (5F10.7)
      END
```

```
      SUBROUTINE COMPUT(Y)
        DOUBLE PRECISION Y

❸ !     Use intrinsic function SIN as an actual argument

        INTRINSIC SIN
        COMMON V(3)

❹ !     Define generic reference to double-precision sine

        V(1) = SIN(Y)

❺ !     Use intrinsic function SIN as an actual argument

        CALL SUB(REAL(Y),SIN)
      END

      SUBROUTINE SUB(A,S)

❻ !     Declare SIN as name of a user function

        EXTERNAL SIN

❶ !     Declare SIN as type DOUBLE PRECISION

        DOUBLE PRECISION SIN
        COMMON V(3)

❷ !     Evaluate intrinsic function SIN

        V(2) = S(A)

❸ !     Evaluate user-defined SIN function

        V(3) = SIN(A)
      END

❹ !     Define the user SIN function

      DOUBLE PRECISION FUNCTION SIN(X)
        INTEGER FACTOR
        SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)     &
            - X**7/FACTOR(7)
      END

      INTEGER FUNCTION FACTOR(N)
        FACTOR = 1
        DO I=N,1,-1
          FACTOR = FACTOR * I
        END DO
      END
```

❶ The statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.

❷ The statement function SIN is called.

❸ The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at ❺.

❹ The generic function name SIN is used to refer to the double-precision sine function.
❺ The single-precision intrinsic sine function is used as an actual argument.
❻ The name SIN is declared a user-defined function name.
❶ The type of SIN is declared double precision.
❷ The single-precision sine function passed at ❺ is evaluated.
❸ The user-defined SIN function is evaluated.
❹ The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

For More Information:

- On the EXTERNAL attribute, see Section 5.8.
- On the scope of names, see Section 2.1.2.
- On the INTRINSIC attribute, see Section 5.11.
- On generic and specific intrinsic functions, see Chapter 9.

8.8.8.2. References to Elemental Intrinsic Procedures

An **elemental intrinsic procedure** has scalar dummy arguments that can be called with scalar or array actual arguments. If actual arguments are array-valued, they must have the same shape. There are many elemental intrinsic functions, but only one elemental intrinsic subroutine (MVBITS).

If the actual arguments are scalar, the result is scalar. If the actual arguments are array-valued, the scalar-valued procedure is applied element-by-element to the actual argument, resulting in an array that has the same shape as the actual argument.

The values of the elements of the resulting array are the same as if the scalar-valued procedure had been applied separately to the corresponding elements of each argument.

For example, if A and B are arrays of shape (5,6), MAX(A, 0.0, B) is an array expression of shape (5,6) whose elements have the value MAX(A (i, j), 0.0, B (i, j)), where i = 1, 2,..., 5, and j = 1, 2,..., 6.

A reference to an elemental intrinsic procedure is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

For More Information:

- On elemental procedures, see Chapter 9.
- On arrays, see Section 3.5.2.

8.8.9. References to Non-Fortran Procedures

To facilitate references to non-Fortran procedures, VSI Fortran provides built-in functions %DESCR, %REF, and %VAL to pass actual arguments; and %LOC, which computes the internal address of a storage item.

8.8.9.1. %DESCR, %REF, and %VAL Argument List Functions

When a procedure is called, Fortran (by default) passes the address of the actual argument, and its length if it is of type character. To call non-Fortran procedures, you may need to pass the actual arguments in a form different from that used by Fortran.

The built-in functions %DESCR, %REF, and %VAL let you change the form of an actual argument. You must specify these functions in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

These functions specify how to pass an actual argument (for example, *a*) to a non-Fortran procedure, as follows:

Function	Effect
%VAL (*a*)	Passes argument *a* as an *n*-bit[1] immediate value. If *a* is integer (or logical) and shorter than *n* bits, it is sign-extended to an *n*-bit value. For complex data types, %VAL passes two *n*-bit arguments.
%REF (*a*)	Passes argument *a* by reference.
%DESCR (*a*)	Passes argument *a* by descriptor.

[1] *n* is 64.

Table 8.1 lists the VSI Fortran defaults for argument passing, and the allowed uses of %DESCR, %REF, and %VAL.

Table 8.1. Defaults for Argument List Functions

Actual Argument Data Type	Default	%VAL	%REF	%DESCR
Expressions:				
Logical	REF	Yes	Yes	Yes
Integer	REF	Yes	Yes	Yes
REAL(4)	REF	Yes	Yes	Yes
REAL(8)	REF	Yes	Yes	Yes
REAL(16)	REF	No	Yes	Yes
COMPLEX(4)	REF	Yes	Yes	Yes
COMPLEX(8)	REF	Yes	Yes	Yes
COMPLEX(16)	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Hollerith	REF	No	No	No
Aggregate[1]	REF	No	Yes	No
Derived	REF	No	Yes	No
Array Name:				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Aggregate [1]	REF	No	Yes	No
Derived	REF	No	Yes	No
Procedure Name:				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes

[1]In VSI Fortran record structures.

The %VAL, %REF, and %DESCR functions override related cDEC$ ATTRIBUTE settings.

For More Information:

On how to use the %VAL, %REF, and %DESCR functions, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.8.9.2. %LOC Function

The built-in function %LOC computes the internal address of a storage item. It takes the following form:

```
%LOC (arg)
```

arg

Is the name of an actual argument. It must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure or statement function.)

The %LOC function produces an integer value that represents the location of the given argument. The value is INTEGER(8). You can use this integer value as an item in an arithmetic expression.

The LOC intrinsic function serves the same purpose as the %LOC built-in function.

For More Information:

- On how to use the %LOC function, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the LOC intrinsic function, see Section 9.4.87.

8.9. Procedure Interfaces

Every procedure has an interface, which consists of the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units.

If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration). The following table shows which procedures have implicit or explicit interfaces:

Kind of Procedure	Interface
External procedure	Implicit[1]
Module Procedure	Explicit
Internal Procedure	Explicit
Intrinsic Procedure	Explicit
Dummy Procedure	Implicit[1]
Statement function	Implicit

[1]Unless an interface block is supplied for the procedure.

The interface of a recursive subroutine or function is explicit within the subprogram that defines it.

An explicit interface can appear in a procedure's definition, in an interface block, or both. (Internal procedures must not appear in an interface block).

The following sections describe when explicit interfaces are required, how to define explicit interfaces, and how to define generic names, operators, and assignment.

8.9.1. Determining When Procedures Require Explicit Interfaces

A procedure must have an explicit interface in the following cases:

- If the procedure has any of the following:
  - An optional dummy argument
  - A dummy argument that is an assumed-shape array, a pointer, or a target
  - A result that is array-valued or a pointer (functions only)
  - A result whose length is neither assumed nor a constant (character functions only)
- If a reference to the procedure appears as follows:
  - With an argument keyword
  - As a reference by its generic name
  - As a defined assignment (subroutines only)
  - In an expression as a defined operator (functions only)
  - In a context that requires it to be pure
- If the procedure is elemental

For More Information:

- On optional arguments, see Section 8.8.1.
- On argument keywords in subroutine references, see Section 7.3.
- On argument keywords in function references, see Section 8.5.2.2.
- On user-defined generic procedures, see Section 8.9.3.
- On defined operators, see Section 8.9.4.
- On defined assignment, see Section 8.9.5.
- On array arguments, see Section 8.8.2.
- On pointer arguments, see Section 8.8.3.
- On pure procedures, see Section 8.5.1.2.
- On elemental procedures, see Section 8.5.1.3.
- On explicit interfaces when calling other languages, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

8.9.2. Defining Explicit Interfaces

Interface blocks define explicit interfaces for external or dummy procedures. They can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines.

An interface block takes the following form:

```
INTERFACE [generic-spec]
  [interface-body]...
  [MODULE PROCEDURE name-list]...
END INTERFACE [generic-spec]
```

`generic-spec`

Is one of the following:

- A generic name
- OPERATOR (op)

  Defines a generic operator (*op*). It can be a defined unary, defined binary, or extended intrinsic operator.
- ASSIGNMENT (=)

  Defines generic assignment.

`interface-body`

Is one or more function or subroutine subprograms. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

The subprogram must not contain a statement function or a DATA, ENTRY, or FORMAT statement; an entry name can be used as a procedure name.

The subprogram can contain a USE statement.

`name-list`

Is the name of one or more module procedures that are accessible in the host. The MODULE PROCEDURE statement is only allowed if the interface block specifies a *generic-spec* and has a host that is a module (or accesses a module by use association).

The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprogram definitions.

Rules and Behavior

Interface blocks can appear in the specification part of the program unit that invokes the external or dummy procedure.

A *generic-spec* can only appear in the END INTERFACE statement (a Fortran 95 feature) if one appears in the INTERFACE statement; they must be identical.

The characteristics specified for the external or dummy procedure must be consistent with those specified in the procedure's definition.

An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

A procedure must not have more than one explicit interface in a given scoping unit.

A interface block containing *generic-spec* specifies a generic interface for the following procedures:

- The procedures within the interface block

  Any generic name, defined operator, or equals symbol that appears is a generic identifier for all the procedures in the interface block. For the rules on how any two procedures with the same generic identifier must differ, see Section 15.3.

- The module procedures listed in the MODULE PROCEDURE statement

  The module procedures must be accessible by a USE statement.

To make an interface block available to multiple program units (through a USE statement), place the interface block in a module.

The following rules apply to interface blocks containing pure procedures:

- The interface specification of a pure procedure must declare the INTENT of all dummy arguments except pointer and procedure arguments.
- A procedure that is declared pure in its definition can also be declared pure in an interface block. However, if it is not declared pure in its definition, it must not be declared pure in an interface block.

Examples

The following example shows a simple procedure interface block with no generic specification:

```
SUBROUTINE SUB_B (B, FB)
  REAL B
  ...
  INTERFACE
    FUNCTION FB (GN)
      REAL FB, GN
    END FUNCTION
  END INTERFACE
```

For More Information:

- On functions, see Section 8.5.2.
- On subroutines, see Section 8.5.3.
- On use and host association, see Section 15.5.1.2.
- On when an explicit interface is required, see Section 8.9.1.
- On when you should not use interface blocks, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On defining generic names, see Section 8.9.3.

- On defining generic operators, see Section 8.9.4.
- On defining generic assignment, see Section 8.9.5.
- On modules, see Section 8.3.
- On pure procedures, see Section 8.5.1.2.

8.9.3. Defining Generic Names for Procedures

An interface block can be used to specify a generic name to reference all of the procedures within the interface block.

The initial line for such an interface block takes the following form:

```
INTERFACE generic-name
```

generic-name

Is the generic name. It can be the same as any of the procedure names in the interface block, or the same as any accessible generic name (including a generic intrinsic name).

This kind of interface block can be used to extend or redefine a generic intrinsic procedure.

The procedures that are given the generic name must be the same kind of subprogram: all must be functions, or all must be subroutines.

Any procedure reference involving a generic procedure name must be resolvable to one specific procedure; it must be unambiguous. For more information, see Section 15.3.

The following is an example of a procedure interface block defining a generic name:

```
INTERFACE GROUP_SUBS
  SUBROUTINE INTEGER_SUB (A, B)
    INTEGER, INTENT(INOUT) :: A, B
  END SUBROUTINE INTEGER_SUB

  SUBROUTINE REAL_SUB (A, B)
    REAL, INTENT(INOUT) :: A, B
  END SUBROUTINE REAL_SUB

  SUBROUTINE COMPLEX_SUB (A, B)
    COMPLEX, INTENT(INOUT) :: A, B
  END SUBROUTINE COMPLEX_SUB
END INTERFACE
```

The three subroutines can be referenced by their individual specific names or by the group name GROUP_SUBS.

The following example shows a reference to INTEGER_SUB:

```
INTEGER V1, V2
CALL GROUP_SUBS (V1, V2)
```

For More Information:

On interface blocks, see Section 8.9.2.

8.9.4. Defining Generic Operators

An interface block can be used to define a generic operator. The only procedures allowed in the interface block are functions that can be referenced as defined operations.

The initial line for such an interface block takes the following form:

```
INTERFACE OPERATOR (op)
```

op

Is one of the following:

- A defined unary operator (one argument)
- A defined binary operator (two arguments)
- An extended intrinsic operator (number of arguments must be consistent with the intrinsic uses of that operator)

The functions within the interface block must have one or two nonoptional arguments with intent IN, and the function result must not be of type character with assumed length. A defined operation is treated as a reference to the function.

The following shows the form (and an example) of a defined unary and defined binary operation:

Operation	Form	Example
Defined Unary	.defined-operator. operand[1]	.MINUS. C
Defined Binary	operand[2] .defined-operator. operand[3]	B .MINUS. C

[1]The operand corresponds to the function's dummy argument.

[2]The left operand corresponds to the first dummy argument of the function.

[3]The right operand corresponds to the second argument.

For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Both forms of each relational operator have the same interpretation, so extending one form (such as >=) defines both forms (>= and .GE.).

The following is an example of a procedure interface block defining a new operator:

```
INTERFACE OPERATOR(.BAR.)
  FUNCTION BAR(A_1)
    INTEGER, INTENT(IN) :: A_1
    INTEGER :: BAR
  END FUNCTION BAR
END INTERFACE
```

The following example shows a way to reference function BAR by using the new operator:

```
INTEGER B
I = 4 + (.BAR. B)
```

The following is an example of a procedure interface block with a defined operator extending an existing operator:

```
INTERFACE OPERATOR(+)
  FUNCTION LGFUNC (A, B)
    LOGICAL, INTENT(IN) :: A(:), B(SIZE(A))
    LOGICAL :: LGFUNC(SIZE(A))
  END FUNCTION LGFUNC
END INTERFACE
```

The following example shows two equivalent ways to reference function LGFUNC:

```
LOGICAL, DIMENSION(1:10) :: C, D, E
N = 10
E = LGFUNC(C(1:N), D(1:N))
E = C(1:N) + D(1:N)
```

For More Information:

- On interface blocks, see Section 8.9.2.
- On intrinsic operators, see Section 4.1.
- On defined operators and operations, see Section 4.1.5.
- On intent, see Section 5.10.

8.9.5. Defining Generic Assignment

An interface block can be used to define generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.

The initial line for such an interface block takes the following form:

```
INTERFACE ASSIGNMENT (=)
```

The subroutines within the interface block must have two nonoptional arguments, the first with intent OUT or INOUT, and the second with intent IN.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see Section 15.3.

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC
```

```
  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                    ! Contains definition of type
 STRING
    TYPE(STRING), INTENT(OUT) :: STR     ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
CALL CHAR_TO_STRING(CH, '432C')
CH = '432C'
```

For More Information:

- On interface blocks, see Section 8.9.2.
- On defined assignment, see Section 4.2.2.
- On intent, see Section 5.10.

8.10. CONTAINS Statement

A CONTAINS statement separates the body of a main program, module, or external subprogram from any internal or module procedures it may contain. It is not executable.

The CONTAINS statement takes the following form:

```
CONTAINS
```

Any number of internal procedures can follow a CONTAINS statement, but a CONTAINS statement cannot appear in the internal procedures themselves.

For More Information:

- On module procedures, see Section 8.3.
- On internal procedures, see Section 8.7.

8.11. ENTRY Statement

The ENTRY statement provides one or more entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram.

The ENTRY statement takes the following form:

```
ENTRY name [([d-arg [,d-arg]...]) [RESULT (r-name)]]
```

name

Is the name of an entry point. If RESULT is specified, this entry name must not appear in any specification statement in the scoping unit of the function subprogram.

d-arg

Is a dummy argument. The dummy argument can be an alternate return indicator (*) if the ENTRY statement is within a subroutine subprogram.

r-name

Is the name of a function result. This name must not be the same as the name of the entry point, or the name of any other function or function result. This parameter can only be specified for function subprograms.

Rules and Behavior

ENTRY statements can only appear in external procedures or module procedures.

An ENTRY statement must not appear in a CASE, DO, IF, FORALL, or WHERE construct, or a nonblock DO loop.

When the ENTRY statement appears in a subroutine subprogram, it is referenced by a CALL statement. When the ENTRY statement appears in a function subprogram, it is referenced by a function reference.

An entry name within a function subprogram can appear in a type declaration statement.

Within the subprogram containing the ENTRY statement, the entry name must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement, and it must not appear in an EXTERNAL or INTRINSIC statement. For example, neither of the following are valid:

```
(1)   SUBROUTINE SUB(E)
      ENTRY E
      ...

(2)   SUBROUTINE SUB
      EXTERNAL E
      ENTRY E
      ...
```

An ENTRY statement can reference itself if the function or subroutine subprogram was defined as RECURSIVE.

Dummy arguments can be used in ENTRY statements even if they differ in order, number, type and kind parameters, and name from the dummy arguments used in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

Dummy arguments can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced. Arguments do not retain their association from one reference of a subprogram to another.

For specific information on ENTRY statements in function subprograms and subroutine subprograms (including examples), see Section 8.11.1 and Section 8.11.2, respectively.

For More Information:

- On functions, see Section 8.5.2.
- On subroutines, see Section 8.5.3.
- On function references, see Section 8.5.2.2.
- On the CALL statement, see Section 7.3.
- On procedure arguments, see Section 8.8.

8.11.1. ENTRY Statements in Function Subprograms

If the ENTRY statement is contained in a function subprogram, it defines an additional function. The name of the function is the name specified in the ENTRY statement, and its result variable is the entry name or the name specified by RESULT (if any).

If the entry result variable has the same characteristics as the FUNCTION statement's result variable, their result variables identify the same variable, even if they have different names. Otherwise, the result variables are storage associated and must all be nonpointer scalars of intrinsic type, in one of the following groups:

Group 1	Type default integer, default real, double precision real, default complex, double complex, or default logical
Group 2	Type REAL (16) and COMPLEX (16)
Group 3	Type default character (with identical lengths)

All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names with the same data type. All associated names with different data types become undefined.

If RESULT is specified in the ENTRY statement and RECURSIVE is specified in the FUNCTION statement, the interface of the function defined by the ENTRY statement is explicit within the function subprogram.

Examples

The following example shows a function subprogram that computes the hyperbolic functions SINH, COSH, and TANH:

```
REAL FUNCTION TANH(X)
  TSINH(Y) = EXP(Y) - EXP(-Y)
  TCOSH(Y) = EXP(Y) + EXP(-Y)

  TANH = TSINH(X)/TCOSH(X)
  RETURN

  ENTRY SINH(X)
  SINH = TSINH(X)/2.0
  RETURN

  ENTRY COSH(X)
  COSH = TCOSH(X)/2.0
```

```
  RETURN
END
```

For More Information:

On the RESULT keyword, see Section 8.5.2.1.

8.11.2. ENTRY Statements in Subroutine Subprograms

If the ENTRY statement is contained in a subroutine subprogram, it defines an additional subroutine. The name of the subroutine is the name specified in the ENTRY statement.

If RECURSIVE is specified on the SUBROUTINE statement, the interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

Examples

The following example shows a main program calling a subroutine containing an ENTRY statement:

```
PROGRAM TEST
  ...
  CALL SUBA(A, B, C)        ! A, B, and C are actual arguments
  ...                       !    passed to entry point SUBA
END
SUBROUTINE SUB(X, Y, Z)
  ...
  ENTRY SUBA(Q, R, S)       ! Q, R, and S are dummy arguments
  ...                       ! Execution starts with this statement
END SUBROUTINE
```

The following example shows an ENTRY statement specifying alternate returns:

```
CALL SUBC(M, N, *100, *200, P)
...
SUBROUTINE SUB(K, *, *)
  ...
  ENTRY SUBC(J, K, *, *, X)
  ...
  RETURN 1
  RETURN 2
END
```

Note that the CALL statement for entry point SUBC includes actual alternate return arguments. The RETURN 1 statement transfers control to statement label 100 and the RETURN 2 statement transfers control to statement label 200 in the calling program.

For More Information:

On implementation of argument association in ENTRY statements, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Chapter 9. Intrinsic Procedures

9.1. Overview of Intrinsic Procedures

Intrinsic procedures are functions and subroutines that are included in the Fortran 95/90 library. There are four classes of intrinsic procedures:

- Elemental procedures

  These procedures have scalar dummy arguments that can be called with scalar or array actual arguments. There are many elemental intrinsic functions and one elemental intrinsic subroutine (MVBITS).

  If the arguments are all scalar, the result is scalar. If an actual argument is array-valued, the intrinsic procedure is applied to each element of the actual argument, resulting in an array that has the same shape as the actual argument.

  If there is more than one array-valued argument, they must all have the same shape.

- Inquiry functions

  These functions have results that depend on the properties of their principal argument, not the value of the argument (the argument value can be undefined).

- Transformational functions

  These functions have one or more array-valued dummy or actual arguments, an array result, or both. The intrinsic function is not applied elementally to an array-valued actual argument; instead it changes (transforms) the argument array into another array.

- Nonelemental procedures

  These procedures must be called with only scalar arguments; they return scalar results. All subroutines (except MVBITS) are nonelemental.

Intrinsic procedures are invoked the same way as other procedures, and follow the same rules of argument association.

The intrinsic procedures have generic (or common) names, and many of the intrinsic functions have specific names. (Some intrinsic functions are both generic and specific.)

In general, generic functions accept arguments of more than one data type; the data type of the result is the same as that of the arguments in the function reference. For elemental functions with more than one argument, all arguments must be of the same type (except for the function MERGE).

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Some specific intrinsic functions are not allowed as actual arguments in all circumstances. Table 9.1 lists specific functions that cannot be passed as actual arguments.

Table 9.1. Functions Not Allowed as Actual Arguments

AIMAX0	EOF	JIDINT	MAX0
AIMIN0	FLOAT	JIFIX	MAX1

AJMAX0	FLOATI	JINT	MIN0
AJMIN0	FLOATJ	JMAX0	MIN1
AKMAX0	FLOATK	JMAX1	MULT_HIGH
AKMIN0	ICHAR	JMIN0	MY_PROCESSOR
AMAX0	IDINT	JMIN1	NUMBER_OF_PROCESSORS
AMAX1	IFIX	KIDINT	NWORKERS
AMIN0	IIDINT	KIFIX	PROCESSORS_SHAPE
AMIN1	IIFIX	KINT	QCMPLX
CHAR	IINT	KIQINT	QEXT
CMPLX	IMAX0	KIQNNT	QEXTD
DBLE	IMAX1	KMAX0	QMAX1
DBLEQ	IMIN0	KMAX1	QMIN1
DCMPLX	IMIN1	KMIN0	QREAL
DFLOTI	INT	KMIN1	RAN
DFLOTJ	INT_PTR_KIND	LGE	REAL
DFLOTK	INT1	LGT	SECNDS
DMAX1	INT2	LLE	SIZEOF
DMIN1	INT4	LLT	SNGL
DPROD	INT8	LOC	SNGLQ
DREAL	JFIX	MALLOC	ZEXT

For More Information:

- On the rules of argument association, see Section 8.8.
- On the MERGE intrinsic function, see Section 9.4.97.
- On optional arguments, see Section 8.8.1.
- On VSI Fortran numeric data format, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On data representation models, see Appendix D.
- On generic intrinsic procedures, see Section 8.8.8.1.
- On elemental references to intrinsic procedures, see Section 8.8.8.2.

9.2. Argument Keywords in Intrinsic Procedures

For all intrinsic procedures, the arguments shown are the names you must use as keywords when using the keyword form for actual arguments. For example, a reference to function CMPLX (X, Y, KIND) can be written as follows:

Using positional arguments:	CMPLX (F, G, L)

Using argument keywords:	CMPLX (KIND=L, Y=G, X=F)[1]

[1]Note that argument keywords can be written in any order.

Some argument keywords are optional (denoted by square brackets). The following describes some of the most commonly used optional arguments:

BACK	Specifies that a string scan is to be in reverse order (right to left).
DIM	Specifies a selected dimension of an array argument.
KIND	Specifies the kind type parameter of the function result.
MASK	Specifies that a mask can be applied to the elements of the argument array to exclude the elements that are not to be involved in an operation.

Examples

The syntax for the DATE_AND_TIME intrinsic subroutine shows four optional positional arguments: DATE, TIME, ZONE, and VALUES (see Section 9.4.35).

The following shows some valid ways to specify these arguments:

```
! Keyword example
CALL DATE_AND_TIME (ZONE=Z)

! The following two positional examples are equivalent
CALL DATE_AND_TIME (DATE, TIME, ZONE)

CALL DATE_AND_TIME (, , ZONE)
```

For More Information:

- On argument keywords in subroutine references, see Section 7.3.
- On argument keywords in function references, see Section 8.5.2.2.
- On argument association, see Section 8.8.

9.3. Categories of Intrinsic Procedures

This section describes the categories of generic intrinsic functions (including a summarizing table), lists the intrinsic subroutines, and provides general information on bit functions.

Intrinsic procedures are fully described (in alphabetical order) in Section 9.4.

9.3.1. Categories of Intrinsic Functions

Generic intrinsic functions can be divided into categories, as shown in Table 9.2.

Table 9.2. Categories of Intrinsic Functions

Category	Subcategory	Description
Numeric	Computation	Perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DPROD, DREAL, FLOAT, FLOOR, IFIX, IMAG, INT, MAX, MAX1, MIN, MIN1, MOD, MODULO,

Category	Subcategory	Description
		NINT, QCMPLX, QEXT, QFLOAT, QREAL, RAN, REAL, SIGN, SNGL, ZEXT
	Manipulation[1]	Return values related to the components of the model values associated with the actual value of the argument: EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING
	Inquiry[1]	Return scalar values from the models associated with the type and kind parameters of their arguments[2]: DIGITS, EPSILON, HUGE, ILEN, MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE, SIZEOF, TINY
	Transformational	Perform vector and matrix multiplication: DOT_PRODUCT, MATMUL
	System	Return information about a process or processor: PROCESSORS_SHAPE, NWORKERS, MY_PROCESSOR, NUMBER_OF_PROCESSORS, SECNDS
Kind type		Return kind type parameters: SELECTED_INT_KIND, SELECTED_REAL_KIND, KIND
Mathematical		Perform mathematical operations: ACOS, ACOSD, ASIN, ASIND, ATAN, ATAND, ATAN2, ATAN2D, COS, COSD, COSH, COTAN, COTAND, EXP, LOG, LOG10, SIN, SIND, SINH, SQRT, TAN, TAND, TANH
Bit	Manipulation	Perform single-bit processing, and logical and shift operations; and allow bit subfields to be referenced: AND, BTEST, IAND, IBCHNG, IBCLR, IBITS, IBSET, IEOR, IOR, ISHA, ISHC, ISHFT, ISHFTC, ISHL, LSHIFT,NOT, OR, RSHIFT, XOR
	Inquiry	Lets you determine parameter *s* (the bit size) in the bit model[3]: BIT_SIZE
	Representation	Return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ
Character	Comparison	Lexically compare character-string arguments and return a default logical result: LGE, LGT, LLE, LLT
	Conversion	Convert character arguments to integer, ASCII, or character values[4]: ACHAR, CHAR, IACHAR, ICHAR
	String handling	Perform operations on character strings, return lengths of arguments, and search for certain arguments: ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, REPEAT, SCAN, TRIM, VERIFY

Category	Subcategory	Description
	Inquiry	Returns length of argument: LEN
Array	Construction	Construct new arrays from the elements of existing array: MERGE, PACK, SPREAD, UNPACK
	Inquiry	Let you determine if an array argument is allocated, and return the size or shape of an array, and the lower and upper bounds of subscripts along each dimension: ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND
	Location	Returns the geometric locations of the maximum and minimum values of an array: MAXLOC, MINLOC
	Manipulation	Let you shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE
	Reduction	Perform operations on arrays. The functions “reduce” elements of a whole array to produce a scalar result, or they can be applied to a specific dimension of an array to produce a result array with a rank reduced by one: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT
Miscellaneous		Do the following: • • Check for pointer association (ASSOCIATED) • Check for end-of-file (EOF) • Return the class of a floating-point argument (FP_CLASS) • Count actual arguments passed to a routine (IARGCOUNT) • Return a pointer to an actual argument list for a routine (IARGPTR) • Return the INTEGER KIND that will hold an address (INT_PTR_KIND) • Test for Not-a-Number values (ISNAN) • Return the internal address of a storage item (LOC) • Return a logical value of an argument (LOGICAL) • Allocate memory (MALLOC)

Category	Subcategory	Description
		• Return the upper 64 bits of a 128-bit unsigned result (MULT_HIGH) • Return a disassociated pointer (NULL) • Check for argument presence (PRESENT) • Convert a bit pattern (TRANSFER)

[1]All of the numeric manipulation, and many of the numeric inquiry functions are defined by the model sets for integers (Section D.1) and reals (Section D.2).

[2]The value of the argument does not have to be defined.

[3]For more information on bit functions, see Section 9.3.3.

[4]The VSI Fortran processor character set is ASCII, so ACHAR = CHAR and IACHAR = ICHAR.

Table 9.3 summarizes the generic intrinsic functions and indicates whether they are elemental, inquiry, or transformational functions, if applicable. Optional arguments are shown within square brackets.

Table 9.3. Summary of Generic Intrinsic Functions

Generic Function	Class	Value Returned
ABS (A)	E	The absolute value of an argument
ACHAR (I)	E	The character in the specified position of the ASCII character set
ACOS (X)	E	The arc cosine (in radians) of the argument
ACOSD (X)	E	The arc cosine (in degrees) of the argument
ADJUSTL (STRING)	E	The specified string with leading blanks removed and placed at the end of the string
ADJUSTR (STRING)	E	The specified string with trailing blanks removed and placed at the beginning of the string
AIMAG (Z)	E	The imaginary part of a complex argument
AINT (A [,KIND])	E	A real value truncated to a whole number
ALL (MASK [,DIM])	T	.TRUE. if all elements of the masked array are true
ALLOCATED (ARRAY)	I	The allocation status of the argument array
AMAX0 (A1, A2 [, A3,...])	E	The maximum value in a list of integers (returned as a real value)
AMIN0 (A1, A2 [, A3,...])	E	The minimum value in a list of integers (returned as a real value)
AND (I, J)	E	See IAND
ANINT (A [,KIND])	E	A real value rounded to a whole number
ANY (MASK [,DIM])	T	.TRUE. if any elements of the masked array are true
ASIN (X)	E	The arc sine (in radians) of the argument
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
ASIND (X)	E	The arc sine (in degrees) of the argument
ASM (STRING [,A,...])	N	A value stored in the appropriate register by the user.
ASSOCIATED (POINTER [,TARGET])	I	.TRUE. if the pointer argument is associated or the pointer is associated with the specified target
ATAN (X)	E	The arc tangent (in radians) of the argument
ATAND (X)	E	The arc tangent (in degrees) of the argument
ATAN2 (Y, X)	E	The inverse arc tangent (in radians) of the arguments
ATAN2D (Y, X)	E	The inverse arc tangent (in degrees) of the arguments
BIT_SIZE (I)	I	Returns the number of bits (*s*) in the bit model
BTEST (I, POS)	E	.TRUE. if the specified position of argument I is one
CEILING (A [,KIND])	E	The smallest integer greater than or equal to the argument value
CHAR (I [,KIND])	E	The character in the specified position of the processor character set
CMPLX (X [,Y] [,KIND])	E	The corresponding complex value of the argument
CONJG (Z)	E	The conjugate of a complex number
COS (X)	E	The cosine of the argument, which is in radians
COSD (X)	E	The cosine of the argument which is in degrees
COSH (X)	E	The hyperbolic cosine of the argument
COTAN (X)	E	The cotangent of the argument, which is in radians
COTAND (X)	E	The cotangent of the argument, which is in degrees
COUNT (MASK [,DIM] [,KIND])	T	The number of .TRUE. elements in the argument array
CSHIFT (ARRAY, SHIFT [,DIM])	T	An array that has the elements of the argument array circularly shifted
DBLE (A)	E	The corresponding double precision value of the argument
DCMPLX (X, Y)	E	The corresponding double complex value of the argument
DFLOAT (A)	E	The corresponding double precision value of the integer argument
DIGITS (X)	I	The number of significant binary digits in the model for the argument
DIM (X, Y)	E	The positive difference between the two arguments
DOT_PRODUCT (VECTOR_A, VECTOR_B)	T	The dot product of two rank-one arrays (also called a vector multiply function)
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])	T	An array that has the elements of the argument array end-off shifted
EPSILON (X)	I	The difference between 1.0 and the next larger model number.
EXP (X)	E	The exponential value for the argument
EXPONENT (X)	E	The value of the exponent part of a real argument
FLOAT (X)	E	The corresponding real value of the integer argument
FLOOR (A [,KIND])	E	The largest integer less than or equal to the argument value
FP_CLASS (X)	E	The class of the IEEE floating-point argument
FRACTION (X)	E	The fractional part of a real argument
HUGE (X)	I	The largest number in the model for the argument
IACHAR (C)	E	The position of the specified character in the ASCII character set
IAND (I, J)	E	The logical AND of the two arguments
IBCHNG (I, POS)	E	The reversed value of a specified bit
IBCLR (I, POS)	E	The specified position of argument I cleared (set to zero)
IBITS (I, POS, LEN)	E	The specified substring of bits of argument I
IBSET (I, POS)	E	The specified bit in argument I set to one
ICHAR (C)	E	The position of the specified character in the processor character set
IEOR (I, J)	E	The logical exclusive OR of the corresponding bit arguments
IFIX (X)	E	The corresponding integer value of the real argument rounded as if it were an implied conversion in an assignment
ILEN (I)	I	The length (in bits) in the two's complement representation of an integer
IMAG (Z)	E	See AIMAG
INDEX (STRING, SUBSTRING [,BACK] [,KIND])	E	The position of the specified substring in a character expression
INT (A [,KIND])	E	The corresponding integer value (truncated) of the argument
IOR (I, J)	E	The logical inclusive OR of the corresponding bit arguments
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
ISHA (I, SHIFT)	E	Argument I shifted left or right by a specified number of bits
ISHC (I, SHIFT)	E	Argument I rotated left or right by a specified number of bits
ISHFT (I, SHIFT)	E	The logical end-off shift of the bits in argument I
ISHFTC (I, SHIFT [,SIZE])	E	The logical circular shift of the bits in argument I
ISHL (I, SHIFT)	E	Argument I logically shifted left or right by a specified number of bits
ISNAN (X)	E	Tests for Not-a-Number (NaN) values
KIND (X)	I	The kind type parameter of the argument
LBOUND (ARRAY [,DIM] [,KIND])	I	The lower bounds of an array (or one of its dimensions)
LEADZ (I)	E	The number of leading zero bits in an integer.
LEN (STRING [,KIND])	I	The length (number of characters) of the argument character string
LEN_TRIM (STRING[,KIND])	E	The length of the specified string without trailing blanks
LGE (STRING_A, STRING_B)	E	A logical value determined by a > or = comparison of the arguments
LGT (STRING_A, STRING_B)	E	A logical value determined by a > comparison of the arguments
LLE (STRING_A, STRING_B)	E	A logical value determined by a < or = comparison of the arguments
LLT (STRING_A, STRING_B)	E	A logical value determined by a < comparison of the arguments
LOC (A)	I	The internal address of the argument.
LOG (X)	E	The natural logarithm of the argument
LOG10 (X)	E	The common logarithm (base 10) of the argument
LOGICAL (L [,KIND])	E	The logical value of the argument converted to a logical of type KIND
LSHIFT (I, POSITIVE_SHIFT)	E	See ISHFT
MATMUL (MATRIX_A, MATRIX_B)	T	The result of matrix multiplication (also called a matrix multiply function)
MAX (A1, A2 [, A3,...])	E	The maximum value in the set of arguments
MAX1 (A1, A2 [, A3,...])	E	The maximum value in the set of real arguments (returned as an integer)
MAXEXPONENT (X)	I	The maximum exponent in the model for the argument
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
MAXLOC (ARRAY [,DIM] [,MASK] [,KIND])	T	The rank-one array that has the location of the maximum element in the argument array
MAXVAL (ARRAY [,DIM] [,MASK])	T	The maximum value of the elements in the argument array
MERGE (TSOURCE, FSOURCE, MASK)	E	An array that is the combination of two conformable arrays (under a mask)
MIN (A1, A2 [, A3,...])	E	The minimum value in the set of arguments
MIN1 (A1, A2 [, A3,...])	E	The minimum value in the set of real arguments (returned as an integer)
MINEXPONENT (X)	I	The minimum exponent in the model for the argument
MINLOC (ARRAY [,DIM] [,MASK] [,KIND])	T	The rank-one array that has the location of the minimum element in the argument array
MINVAL (ARRAY [,DIM] [,MASK])	T	The minimum value of the elements in the argument array
MOD (A, P)	E	The remainder of the arguments (has the sign of the first argument)
MODULO (A, P)	E	The modulo of the arguments (has the sign of the second argument)
NEAREST (X, S)	E	The nearest different machine-representable number in a given direction
NINT (A [,KIND])	E	A real value rounded to the nearest integer
NOT (I)	E	The logical complement of the argument
NULL ([MOLD])	T	A disassociated pointer
OR (I, J)	E	See IOR
PACK (ARRAY, MASK [,VECTOR])	T	A packed array of rank one (under a mask)
POPCNT (I)	E	The number of 1 bits in an integer.
POPPAR (I)	E	The parity of an integer.
PRECISION (X)	I	The decimal precision (real or complex) of the argument
PRESENT (A)	I	.TRUE. if an actual argument has been provided for an optional dummy argument
PRODUCT (ARRAY [,DIM] [,MASK])	T	The product of the elements of the argument array
QCMPLX (X, Y)	E	The corresponding COMPLEX(16) value of the argument
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
QEXT (A)	E	The corresponding REAL(16) precision value of the argument.
QFLOAT (A)	E	The corresponding REAL(16) precision value of the integer argument.
RADIX (X)	I	The base of the model for the argument
RANGE (X)	I	The decimal exponent range of the model for the argument
REAL (A [,KIND])	E	The corresponding real value of the argument
REPEAT (STRING, NCOPIES)	T	The concatenation of zero or more copies of the specified string
RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])	T	An array that has a different shape than the argument array, but the same elements
RRSPACING (X)	E	The reciprocal of the relative spacing near the argument
RSHIFT (I, NEGATIVE_SHIFT)	E	See ISHFT
SCALE (X, I)	E	The value of the exponent part (of the model for the argument) changed by a specified value
SCAN (STRING, SET [,BACK] [,KIND])	E	The position of the specified character (or set of characters) within a string
SELECTED_INT_KIND (R)	T	The integer kind parameter of the argument
SELECTED_REAL_KIND ([P] [,R])	T	The real kind parameter of the argument; one of the optional arguments must be specified
SET_EXPONENT (X, I)	E	The value the first argument would have if its exponent part were set to the second argument
SHAPE (SOURCE [,KIND])	I	The shape (rank and extents) of an array or scalar
SIGN (A, B)	E	A value with the sign transferred from its second argument
SIN (X)	E	The sine of the argument, which is in radians
SIND (X)	E	The sine of the argument, which is in degrees
SINH (X)	E	The hyperbolic sine of the argument
SIZE (ARRAY [,DIM] [,KIND])	I	The size (total number of elements) of the argument array (or one of its dimensions)
SNGL (X)	E	The corresponding real value of the argument
SPACING (X)	E	The value of the absolute spacing of model numbers near the argument
SPREAD (SOURCE, DIM, NCOPIES)	T	A replicated array that has an added dimension
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Generic Function	Class	Value Returned
SQRT (X)	E	The square root of the argument
SUM (ARRAY [,DIM] [,MASK])	T	The sum of the elements of the argument array
TAN (X)	E	The tangent of the argument, which is in radians
TAND (X)	E	The tangent of the argument, which is in degrees
TANH (X)	E	The hyperbolic tangent of the argument
TINY (X)	I	The smallest positive number in the model for the argument
TRAILZ (I)	E	The number of trailing zero bits in an integer.
TRANSFER (SOURCE, MOLD [,SIZE])	T	The bit pattern of SOURCE converted to the type and kind parameters of MOLD
TRANSPOSE (MATRIX)	T	The matrix transpose for the rank-two argument array
TRIM (STRING)	T	The argument with trailing blanks removed
UBOUND (ARRAY [,DIM] [,KIND])	I	The upper bounds of an array (or one of its dimensions)
UNPACK (VECTOR, MASK, FIELD)	T	An array (under a mask) unpacked from a rank-one array
VERIFY (STRING, SET [,BACK] [,KIND])	E	The position of the first character in a string that does not appear in the given set of characters
XOR (I, J)	E	See IEOR
ZEXT (X [,KIND])	E	A zero-extended value of the argument
Key to Classes **E–Elemental** **I–Inquiry** **T–Transformational** **N–Nonelemental**		

Table 9.4 lists the specific functions that have no generic function associated with them.

Table 9.4. Specific Functions with No Generic Association

Specific Function	Class	Value Returned
DPROD (X, Y)	E	The higher precision product of two real arguments
DREAL (A)	E	The corresponding double-precision value of the real part of a double-complex argument
EOF (A)	I	.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record
MALLOC (I)	E	The starting address for the block of memory allocated
MULT_HIGH (I, J)	E	The upper (leftmost) 64 bits of the 128-bit unsigned result.
Key to Classes **E–Elemental** **I–Inquiry** **N–Nonelemental**		

Specific Function	Class	Value Returned
NUMBER_OF_PROCESSORS ([DIM])	I	The total number of processors (peers) available to the program
MY_PROCESSOR ( )	I	The identifying number of the calling process
NWORKERS ( )[1]	I	The number of executing processes
PROCESSORS_SHAPE ( )	I	The shape of an implementation-dependent hardware processor array
QREAL (A)	E	The corresponding REAL(16) value of the real part of a COMPLEX(16) argument
RAN (I)	N	The next number from a sequence of pseudorandom numbers (uniformly distributed in the range 0 to 1)
SECNDS (X)	E	The system time of day (or elapsed time) as a floating-point value in seconds
SIZEOF (X)	I	The bytes of storage used by the argument
Key to Classes **E–Elemental** **I–Inquiry** **N–Nonelemental**		

[1]Included for compatibility with older versions of Fortran 77.

9.3.2. Intrinsic Subroutines

Table 9.5 lists the intrinsic subroutines. All these subroutines are nonelemental except for MVBITS.

Table 9.5. Intrinsic Subroutines

Subroutine	Value Returned or Result
CPU_TIME (TIME)	The processor time in seconds
DATE (BUF)	The ASCII representation of the current date (in dd-mmm-yy form)
DATE_AND_TIME ([DATE] [,TIME] [,ZONE] [,VALUES])	Date and time information from the real-time clock
ERRSNS ([IO_ERR] [,SYS_ERR] [,STAT] [,UNIT] [,COND])	Information about the most recently detected error condition
EXIT ([STATUS])	Optionally returns image exit status; terminates the program, closes all files, and returns control to the operating system
FREE (A)	Frees memory that is currently allocated
IDATE (I, J, K)	Three integer values representing the current month, day, and year
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)[1]	Copies a sequence of bits (bit field) from one location to another
RANDOM_NUMBER (HARVEST)	A pseudorandom number taken from a sequence of pseudorandom numbers uniformly distributed within the range $0 <= x < 1$
RANDOM_SEED ([SIZE] [,PUT] [,GET])	Initializes or retrieves the pseudorandom number generator seed value

Subroutine	Value Returned or Result
RANDU (I1, I2, X)	A pseudorandom number as a single-precision value (within the range 0.0 to 1.0)
SYSTEM_CLOCK ([COUNT] [,COUNT_RATE] [,COUNT_MAX])	Data from the processors real-time clock
TIME (BUF)	The ASCII representation of the current time (in hh:mm:ss form)

[1] An elemental subroutine.

9.3.3. Bit Functions

Integer data types are represented internally in binary twos complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0.

The intrinsic functions IAND, IOR, IEOR, and NOT operate on all of the bits of their argument (or arguments). Bit 0 of the result comes from applying the specified logical operation to bit 0 of the argument. Bit 1 of the result comes from applying the specified logical operation to bit 1 of the argument, and so on for all of the bits of the result.

The functions ISHFT and ISHFTC shift binary patterns.

The functions IBSET, IBCLR, BTEST, and IBITS and the subroutine MVBITS operate on bit fields.

A **bit field** is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 47 is represented by the following:

Binary pattern:	0...0101111
Bit position:	n...6543210
	Where *n* is the number of bit positions in the numeric storage unit.

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4.

Negative integers are represented in twos complement notation. For example, the integer –47 is represented by the following:

Binary pattern:	1...1010001
Bit position:	n...6543210
	Where n is the number of bit positions in the numeric storage unit.

The value of bit position n is as follows:

1 for a negative number
0 for a non-negative number

All the high-order bits in the pattern from the last significant bit of the value up to bit n are the same as bit n.

IBITS and MVBITS operate on general bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate on 1-bit fields. They do not require a length argument.

For IBSET, IBCLR, and BTEST, the bit position range is as follows:

- 0 to 63 for INTEGER(8) and LOGICAL(8)
- 0 to 31 for INTEGER(4) and LOGICAL(4)
- 0 to 15 for INTEGER(2) and LOGICAL(2)
- 0 to 7 for BYTE, INTEGER(1), and LOGICAL(1)

For IBITS, the bit position can be any number. The length range is 0 to 63.

The following example demonstrates IBSET, IBCLR, and BTEST:

```
I = 4
J = IBSET (I,5)
PRINT *, 'J = ',J
K = IBCLR (J,2)
PRINT *, 'K = ',K
PRINT *, 'Bit 2 of K is ',BTEST(K,2)
END
```

The results are: J = 36, K = 32, and Bit 2 of K is F.

For optimum selection of performance and memory requirements, VSI Fortran provides the following integer data types:

Data Type	**Storage Required (in bytes)**
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8

The bit manipulation functions each have a generic form that operates on all of these integer types and a specific form for each type.

When you specify the intrinsic functions that refer to bit positions or that shift binary patterns within a storage unit, be careful that you do not create a value that is outside the range of integers representable by the data type. If you shift by an amount greater than or equal to the size of the object you're shifting, the result is 0.

Consider the following:

```
INTEGER(2) I,J
I = 1
J = 17
I = ISHFT(I,J)
```

The variables I and J have INTEGER(2) type. Therefore, the generic function ISHFT maps to the specific function IISHFT, which returns an INTEGER(2) result. INTEGER(2) results must be in the range –32768 to 32767, but the value 1, shifted left 17 positions, yields the binary pattern 1 followed by 17 zeros, which represents the integer 131072. In this case, the result in I is 0.

The previous example would be valid if I was INTEGER(4), because ISHFT would then map to the specific function JISHFT, which returns an INTEGER(4) value.

If ISHFT is called with a constant first argument, the result will either be the default integer size or the smallest integer size that can contain the first argument, whichever is larger.

9.4. Descriptions of Intrinsic Procedures

This section contains detailed information on all the generic and specific intrinsic procedures. These procedures are described in alphabetical order by generic name (if there is one). In headings, square brackets denote optional arguments; in text, these optional arguments are labeled “(opt)”.

9.4.1. ABS (A)

Description: Computes an absolute value.

Class: Elemental function; Generic

Arguments: A must be of type integer, real, or complex.

Results: If A is an integer or real value, the value of the result is |A |; if A is a complex value (X, Y), the result is the real value SQRT (X**2 + Y**2).

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIABS	INTEGER(2)	INTEGER(2)
IABS[1]	INTEGER(4)	INTEGER(4)
KIABS	INTEGER(8)	INTEGER(8)
ABS	REAL(4)	REAL(4)
DABS	REAL(8)	REAL(8)
QABS	REAL(16)	REAL(16)
CABS[2]	COMPLEX(4)	REAL(4)
CDABS[3]	COMPLEX(8)	REAL(8)
CQABS	COMPLEX(16)	REAL(16)

[1]Or JIABS. For compatibility with older versions of Fortran, IABS can also be specified as a generic function.

[2]The setting of compiler options specifying real size can affect CABS.

[3]This function can also be specified as ZABS.

Examples

ABS (–7.4) has the value 7.4.

ABS ((6.0, 8.0)) has the value 10.0.

9.4.2. ACHAR (I)

Description: Returns the character in a specified position of the ASCII character set, even if the processor's default character set is different. It is the inverse of the IACHAR function. In VSI Fortran, ACHAR is equivalent to the CHAR function.

Class: Elemental function; Generic

Arguments: I must be of type integer.

Results: The result is of type character with length 1; it has the kind parameter value of KIND ('A').

If I has a value within the range 0 to 127, the result is the character in position I of the ASCII character set. ACHAR (IACHAR(C)) has the value C for any character C capable of representation in the processor.

Examples

ACHAR (71) has the value 'G'.

ACHAR (63) has the value '?'.

9.4.3. ACOS (X)

Description: Produces the arccosine of X.

Class: Elemental function; Generic

Arguments: X must be of type real. The |X | must be less than or equal to 1.

Results: The result type is the same as X and is expressed in radians. The value lies in the range 0 to π.

Specific Name	Argument Type	Result Type
ACOS	REAL(4)	REAL(4)
DACOS	REAL(8)	REAL(8)
QACOS	REAL(16)	REAL(16)

Examples

ACOS (0.68032123) has the value .8225955.

9.4.4. ACOSD (X)

Description: Produces the arccosine of X.

Class: Elemental function; Generic

Arguments: X must be of type real and must be greater than or equal to zero. The |X | must be less than or equal to 1.

Results: The result type is the same as X and is expressed in degrees.

Specific Name	Argument Type	Result Type
ACOSD	REAL(4)	REAL(4)
DACOSD	REAL(8)	REAL(8)
QACOSD	REAL(16)	REAL(16)

Examples

ACOSD (0.886579) has the value 27.55354.

9.4.5. ADJUSTL (STRING)

Description:	Adjusts a character string to the left, removing leading blanks and inserting trailing blanks.
Class:	Elemental function; Generic
Arguments:	STRING must be of type character.
Results:	The result is of type character with the same length and kind parameter as STRING. The value of the result is the same as STRING, except that any leading blanks have been removed and inserted as trailing blanks.

Examples

ADJUSTL ('ΔΔΔΔSUMMERTIME') has the value 'SUMMERTIMEΔΔΔΔ'.

9.4.6. ADJUSTR (STRING)

Description:	Adjusts a character string to the right, removing trailing blanks and inserting leading blanks.
Class:	Elemental function; Generic
Arguments:	STRING must be of type character.
Results:	The result is of type character with the same length and kind parameter as STRING. The value of the result is the same as STRING, except that any trailing blanks have been removed and inserted as leading blanks.

Examples

ADJUSTR ('SUMMERTIMEΔΔΔΔ') has the value 'ΔΔΔΔSUMMERTIME'.

9.4.7. AIMAG (Z)

Description:	Returns the imaginary part of a complex number.[1]
Class:	Elemental function; Generic
Arguments:	Z must be of type complex.
Results:	The result is of type real with the same kind parameter as Z. If Z has the value (x, y), the result has the value *y*.

[1]This function can also be specified as IMAG.

Specific Name	Argument Type	Result Type
AIMAG[1]	COMPLEX(4)	REAL(4)
DIMAG	COMPLEX(8)	REAL(8)
QIMAG	COMPLEX(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect AIMAG.

Examples

AIMAG ((4.0, 5.0)) has the value 5.0.

9.4.8. AINT (A [,KIND])

Description: Truncates a value to a whole number.

Class: Elemental function; Generic

Arguments:

A	Must be of type real.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type real. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter is that of A.

The result is defined as the largest integer whose magnitude does not exceed the magnitude of A and whose sign is the same as that of A. If |A | is less than 1, AINT (A) has the value zero.

Specific Name	Argument Type	Result Type
AINT	REAL(4)	REAL(4)
DINT	REAL(8)	REAL(8)
QINT	REAL(16)	REAL(16)

Examples

AINT (3.678) has the value 3.0.

AINT (–1.375) has the value –1.0.

9.4.9. ALL (MASK [,DIM])

Description: Determines if *all* values are true in an entire array or in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

MASK	Must be a logical array.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of MASK.

Results: The result is an array or a scalar of type logical.

The result is a scalar if DIM is omitted or MASK has rank one. A scalar result is true only if all elements of MASK are true, or MASK has size zero. The result has the value false if any element of MASK is false.

An array result has the same type and kind parameters as MASK, and a rank that is one less than MASK. Its shape is (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of MASK.

Each element in an array result is true only if all elements in the one dimensional array defined by MASK (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) are true.

Examples

ALL ((/.TRUE., .FALSE., .TRUE./)) has the value false because some elements of MASK are not true.

ALL ((/.TRUE., .TRUE., .TRUE./)) has the value true because *all* elements of MASK are true.

A is the array $\begin{bmatrix} 1 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ and B is the array $\begin{bmatrix} 0 & 5 & 7 \\ 2 & 6 & 9 \end{bmatrix}$.

ALL (A .EQ. B, DIM=1) tests to see if all elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, false) because only the second column has elements that are all equal.

ALL (A .EQ. B, DIM=2) tests to see if all elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (false, false) because each row has some elements that are not equal.

9.4.10. ALLOCATED (ARRAY)

Description:	Indicates whether an allocatable array is currently allocated.
Class:	Inquiry function; Generic
Arguments:	ARRAY must be an allocatable array.
Results:	The result is a scalar of type default logical. The result has the value true if ARRAY is currently allocated, false if ARRAY is not currently allocated, or undefined if its allocation status is undefined.

Examples

Consider the following:

```
REAL, ALLOCATABLE, DIMENSION (:,:,:) :: E
PRINT *, ALLOCATED (E)       ! Returns the value false
ALLOCATE (E (12, 15, 20))
PRINT *, ALLOCATED (E)       ! Returns the value true
```

9.4.11. ANINT (A [,KIND])

Description:	Calculates the nearest whole number.	
Class:	Elemental function; Generic	
Arguments:	A	Must be of type real.
	KIND (opt)	Must be a scalar integer initialization expression.
Results:	The result is of type real. If KIND is present, the kind parameter is that specified by KIND; otherwise, the kind parameter is that of A. If A is greater than zero, ANINT (A) has the value AINT (A + 0.5); if A is less than or equal to zero, ANINT (A) has the value AINT (A – 0.5).	

Specific Name	Argument Type	Result Type
ANINT	REAL(4)	REAL(4)
DNINT	REAL(8)	REAL(8)
QNINT	REAL(16)	REAL(16)

Examples

ANINT (3.456) has the value 3.0.

ANINT (–2.798) has the value –3.0.

9.4.12. ANY (MASK [,DIM])

Description:	Determines if *any* value is true in an entire array or in a specified dimension of an array.	
Class:	Transformational function; Generic	
Arguments:	MASK	Must be a logical array.
	DIM (opt)	Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of MASK.
Results:	The result is an array or a scalar of type logical.	

The result is a scalar if DIM is omitted or MASK has rank one. A scalar result is true if any elements of MASK are true. The result has the value false if no element of MASK is true, or MASK has size zero.

An array result has the same type and kind parameters as MASK, and a rank that is one less than MASK. Its shape is (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of MASK.

Each element in an array result is true if any elements in the one dimensional array defined by MASK (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) are true.

Examples

ANY ((/.FALSE., .FALSE., .TRUE./)) has the value true because one element is true.

A is the array $\begin{bmatrix} 1 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ and B is the array $\begin{bmatrix} 0 & 5 & 7 \\ 2 & 6 & 9 \end{bmatrix}$.

ANY (A .EQ. B, DIM=1) tests to see if *any* elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, true) because the second and third columns have at least one element that is equal.

ANY (A .EQ. B, DIM=2) tests to see if any elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (true, true) because each row has at least one element that is equal.

9.4.13. ASIN (X)

Description:	Produces the arcsine of X.
Class:	Elemental function; Generic
Arguments:	X must be of type real. The |X | must be less than or equal to 1.
Results:	The result type is the same as X and is expressed in radians. The value lies in the range -- π/2 to π/2.

Specific Name	Argument Type	Result Type
ASIN	REAL(4)	REAL(4)
DASIN	REAL(8)	REAL(8)
QASIN	REAL(16)	REAL(16)

Examples

ASIN (0.79345021) has the value 0.9164571.

9.4.14. ASIND (X)

Description: Produces the arcsine of X.

Class: Elemental function; Generic

Arguments: X must be of type real and must be greater than or equal to zero. The |X | must be less than or equal to 1.

Results: The result type is the same as X and is expressed in degrees.

Specific Name	Argument Type	Result Type
ASIND	REAL(4)	REAL(4)
DASIND	REAL(8)	REAL(8)
QASIND	REAL(16)	REAL(16)

Examples

ASIND (0.2467590) has the value 14.28581.

9.4.15. ASSOCIATED (POINTER [,TARGET])

Description: Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.

Class: Inquiry function; Generic

Arguments:

POINTER	Must be a pointer (of any data type).
TARGET (opt)	Must be a pointer or target.

The pointer (in POINTER or TARGET) must not have an association status that is undefined.

Results: The result is a scalar of type default logical.

If only POINTER appears, the result is true if it is currently associated with a target; otherwise, the result is false.

If TARGET also appears and is a target, the result is true if POINTER is currently associated with TARGET; otherwise, the result is false.

If TARGET is a pointer, the result is true if both POINTER and TARGET are currently associated with the same target; otherwise, the result is false. (If either POINTER or TARGET is disassociated, the result is false.)

The setting of compiler options specifying integer size can affect this function.

Examples

Consider the following:

```
REAL, TARGET, DIMENSION (0:50) :: TAR
REAL, POINTER, DIMENSION (:) :: PTR
PTR => TAR
PRINT *, ASSOCIATED (PTR, TAR)     ! Returns the value true
```

The subscript range for PTR is 0:50. Consider the following pointer assignment statements:

```
(1) PTR => TAR (:)
(2) PTR => TAR (0:50)
(3) PTR => TAR (0:49)
```

For statements 1 and 2, ASSOCIATED (PTR, TAR) is true because TAR has not changed (the subscript range for PTR in both cases is 1:51, following the rules for deferred-shape arrays). For statement 3, ASSOCIATED (PTR, TAR) is false because the upper bound of PTR has changed.

Consider the following:

```
REAL, POINTER, DIMENSION (:) :: PTR2, PTR3
ALLOCATE (PTR2 (0:15))
PTR3 => PTR2
PRINT *, ASSOCIATED (PTR2, PTR3)   ! Returns the value true
...
NULLIFY (PTR2)
NULLIFY (PTR3)
PRINT *, ASSOCIATED (PTR2, PTR3)   ! Returns the value false
```

9.4.16. ATAN (X)

Description: Produces the arctangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X and is expressed in radians. The value lies in the range -- π/2 to π/2.

Specific Name	Argument Type	Result Type
ATAN	REAL(4)	REAL(4)
DATAN	REAL(8)	REAL(8)
QATAN	REAL(16)	REAL(16)

Examples

ATAN (1.5874993) has the value 1.008666.

9.4.17. ATAND (X)

Description: Produces the arctangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real and must be greater than or equal to zero.

Results: The result type is the same as X and is expressed in radians.

Specific Name	Argument Type	Result Type
ATAND	REAL(4)	REAL(4)
DATAND	REAL(8)	REAL(8)
QATAND	REAL(16)	REAL(16)

Examples

ATAND (0.0874679) has the value 4.998819.

9.4.18. ATAN2 (Y, X)

Description:	Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (X, Y).	
Class:	Elemental function; Generic	
Arguments:	Y	Must be of type real.
	X	Must have the same type and kind parameters as Y. If Y has the value zero, X cannot have the value zero.
Results:	The result type is the same as X and is expressed in radians. The value lies in the range -π < ATAN2 (Y,X) <= π. If X /= zero, the result is approximately equal to the value of arctan (Y/X). If Y > zero, the result is positive. If Y < zero, the result is negative. If Y = zero, the result is zero (if X > zero) or π (if X < zero). If X = zero, the absolute value of the result is π/2.	

Specific Name	Argument Type	Result Type
ATAN2	REAL(4)	REAL(4)
DATAN2	REAL(8)	REAL(8)
QATAN2	REAL(16)	REAL(16)

Examples

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, then ATAN2 (Y, X) is $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ \frac{-3\pi}{4} & \frac{-\pi}{4} \end{bmatrix}$.

9.4.19. ATAN2D (Y, X)

Description:	Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (X, Y).	
Class:	Elemental function; Generic	
Arguments:[1]	Y	Must be of type real.
	X	Must have the same type and kind parameters as Y.
Results:	The result type is the same as X and is expressed in degrees. The value lies in the range –180 degrees to 180 degrees. If X /= zero, the result is approximately equal to the value of arctan (Y/X). If Y > zero, the result is positive.	

If Y < zero, the result is negative.

If Y = zero, the result is zero (if X > zero) or 180 degrees (if X < zero).

If X = zero, the absolute value of the result is 90 degrees.

[1]Both arguments must not have the value zero.

Specific Name	Argument Type	Result Type
ATAN2D	REAL(4)	REAL(4)
DATAN2D	REAL(8)	REAL(8)
QATAN2D	REAL(16)	REAL(16)

Examples

ATAN2D (2.679676, 1.0) has the value 69.53546.

9.4.20. BIT_SIZE (I)

Description: Returns the number of bits in an integer type.

Class: Inquiry function; Generic

Arguments: I must be of type integer.

Results: The result is a scalar integer with the same kind parameter as I. The result value is the number of bits (*s*) defined by the bit model for integers with the kind parameter of the argument. For information on the bit model, see Section D.3.

Examples

BIT_SIZE (1_2) has the value 16 because the KIND=2 integer type contains 16 bits.

9.4.21. BTEST (I, POS)

Description: Tests a bit of an integer argument.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
POS	Must be of type integer. It must not be negative and it must be less than BIT_SIZE (I). The rightmost (least significant) bit of I is in position 0.

Results: The result is of type default logical.

The result is true if bit POS of I has the value 1. The result is false if POS has the value zero. For more information on bit functions, see Section 9.3.3.

For information on the model for the interpretation of an integer value as a sequence of bits, see Section D.3.

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument Type	Result Type
	INTEGER(1)	LOGICAL(1)

Specific Name	Argument Type	Result Type
BITEST	INTEGER(2)	LOGICAL(2)
BTEST[1]	INTEGER(4)	LOGICAL(4)
BKTEST	INTEGER(8)	LOGICAL(8)

[1]Or BJTEST

Examples

BTEST (9, 3) has the value true.

If A has the value $\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$, the value of BTEST (A, 2) is $\begin{bmatrix}\texttt{false} & \texttt{false}\\\texttt{false} & \texttt{true}\end{bmatrix}$ and the value of BTEST (2, A) is $\begin{bmatrix}\texttt{true} & \texttt{false}\\\texttt{false} & \texttt{false}\end{bmatrix}$.

9.4.22. CEILING (A [,KIND])

Description: Returns the smallest integer greater than or equal to its argument.

Class: Elemental function; Generic

Arguments:

A	Must be of type real.
KIND (opt)	Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the smallest integer greater than or equal to A.

Examples

CEILING (4.8) has the value 5.

CEILING (–2.55) has the value –2.0.

9.4.23. CHAR (I [,KIND])

Description: Returns the character in the specified position of the processor's character set. It is the inverse of the function ICHAR.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer with a value in the range 0 to n – 1, where *n* is the number of characters in the processor's character set.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type character with length 1. The kind parameter is that of default character type.

The result is the character in position I of the processor's character set. ICHAR(CHAR (I, KIND(C))) has the value I for 0 to n – 1 and CHAR(ICHAR(C), KIND(C)) has the value C for any character C capable of representation in the processor.

Specific Name	Argument Type	Result Type
	INTEGER(1)	CHARACTER
	INTEGER(2)	CHARACTER
CHAR[1]	INTEGER(4)	CHARACTER
	INTEGER(8)	CHARACTER

[1]This specific function cannot be passed as an actual argument.

Examples

CHAR (76) has the value 'L'.

CHAR (94) has the value '^'.

9.4.24. CMPLX (X [,Y] [,KIND])

Description:	Converts an argument to complex type. This function must not be passed as an actual argument.	
Class:	Elemental function; Generic	
Arguments:	X	Must be of type integer, real, or complex.
	Y (opt)	Must be of type integer or real. It must not be present if X is of type complex.
	KIND (opt)	Must be a scalar integer initialization expression.
Results:	The result is of type complex (COMPLEX(4) or COMPLEX*8). If KIND is present, the kind parameter is that specified by KIND; otherwise, the kind parameter is that of default real type. If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If Y is not specified and X is complex, the result value is CMPLX (REAL(X), AIMAG(X)). If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part. CMPLX(X, Y, KIND) has the complex value whose real part is REAL(X, KIND) and whose imaginary part is REAL(Y, KIND). The setting of compiler options specifying real size can affect this function.	

Examples

CMPLX (–3) has the value (–3.0, 0.0).

CMPLX (4.1, 2.3) has the value (4.1, 2.3).

9.4.25. CONJG (Z)

Description:	Calculates the conjugate of a complex number.
Class:	Elemental function; Generic
Arguments:	Z must be of type complex.
Results:	The result type is the same as Z. If Z has the value (x, y), the result has the value (x, –y).

Specific Name	Argument Type	Result Type
CONJG	COMPLEX(4)	COMPLEX(4)
DCONJG	COMPLEX(8)	COMPLEX(8)
QCONJG	COMPLEX(16)	COMPLEX(16)

Examples

CONJG ((2.0, 3.0)) has the value (2.0, –3.0).

CONJG ((1.0, –4.2)) has the value (1.0, 4.2).

9.4.26. COS (X)

Description: Produces the cosine of X.

Class: Elemental function; Generic

Arguments: X must be of type real or complex. It must be in radians and is treated as modulo 2* π. (If X is of type complex, its real part is regarded as a value in radians.)

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
COS	REAL(4)	REAL(4)
DCOS	REAL(8)	REAL(8)
QCOS	REAL(16)	REAL(16)
CCOS[1]	COMPLEX(4)	COMPLEX(4)
CDCOS[2]	COMPLEX(8)	COMPLEX(8)
CQCOS	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CCOS.

[2]This function can also be specified as ZCOS.

Examples

COS (2.0) has the value –0.4161468.

COS (0.567745) has the value 0.8431157.

9.4.27. COSD (X)

Description: Produces the cosine of X.

Class: Elemental function; Generic

Arguments: X must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
COSD	REAL(4)	REAL(4)
DCOSD	REAL(8)	REAL(8)
QCOSD	REAL(16)	REAL(16)

Examples

COSD (2.0) has the value 0.9993908.

COSD (30.4) has the value 0.8625137.

9.4.28. COSH (X)

Description: Produces a hyperbolic cosine.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
COSH	REAL(4)	REAL(4)
DCOSH	REAL(8)	REAL(8)
QCOSH	REAL(16)	REAL(16)

Examples

COSH (2.0) has the value 3.762196.

COSH (0.65893) has the value 1.225064.

9.4.29. COTAN (X)

Description: Produces the cotangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real; it cannot be zero. It must be in radians and is treated as modulo 2* π.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
COTAN	REAL(4)	REAL(4)
DCOTAN	REAL(8)	REAL(8)
QCOTAN	REAL(16)	REAL(16)

Examples

COTAN (2.0) has the value –4.576575E–01.

COTAN (0.6) has the value 1.461696.

9.4.30. COTAND (X)

Description: Produces the cotangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
COTAND	REAL(4)	REAL(4)
DCOTAND	REAL(8)	REAL(8)

Specific Name	Argument Type	Result Type
QCOTAND	REAL(16)	REAL(16)

Examples

COTAND (2.0) has the value 0.2863625E+02.

COTAND (0.6) has the value 0.9548947E+02.

9.4.31. COUNT (MASK [,DIM] [,KIND])

Description: Counts the number of true elements in an entire array or in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

MASK	Must be a logical array.
DIM (opt)	Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of MASK.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is an array or a scalar of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is a scalar if DIM is omitted or MASK has rank one. A scalar result has a value equal to the number of true elements of MASK. If MASK has size zero, the result is zero.

An array result has a rank that is one less than MASK, and shape ($d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n$), where ($d_1, d_2,..., d_n$) is the shape of MASK.

Each element in an array result equals the number of elements that are true in the one dimensional array defined by MASK ($s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$).

Examples

COUNT ((/.TRUE., .FALSE., .TRUE./)) has the value 2 because two elements are true.

COUNT ((/.TRUE., .TRUE., .TRUE./)) has the value 3 because three elements are true.

A is the array $\begin{bmatrix}1 & 5 & 7\\3 & 6 & 8\end{bmatrix}$ and B is the array $\begin{bmatrix}0 & 5 & 7\\2 & 6 & 9\end{bmatrix}$.

COUNT (A .NE. B, DIM=1) tests to see how many elements in each column of A are not equal to the elements in the corresponding column of B. The result has the value (2, 0, 1) because:

- The first column of A and B have 2 elements that are not equal.
- The second column of A and B have 0 elements that are not equal.
- The third column of A and B have 1 element that is not equal.

COUNT (A .NE. B, DIM=2) tests to see how many elements in each row of A are not equal to the elements in the corresponding row of B. The result has the value (1, 2) because:

- The first row of A and B have 1 element that is not equal.
- The second row of A and B have 2 elements that are not equal.

9.4.32. CPU_TIME (TIME)

Description: Returns a processor-dependent approximation of the processor time in seconds. This is a new intrinsic procedure in Fortran 95.

Class: Subroutine

Arguments: TIME must be scalar and of type real. It is an INTENT(OUT) argument.

If a meaningful time cannot be returned, a processor-dependent negative value is returned.

Examples

Consider the following:

```
REAL time_begin, time_end
...
CALL CPU_TIME(time_begin)
...
CALL CPU_TIME(time_end)

PRINT (*,*) 'Time of operation was ', time_end - time_begin, ' seconds'
```

9.4.33. CSHIFT (ARRAY, SHIFT [,DIM])

Description: Performs a circular shift on a rank-one array, or performs circular shifts on all the complete rank-one sections (vectors) along a given dimension of an array of rank two or greater.

Elements shifted off one end are inserted at the other end. Different sections can be shifted by different amounts and in different directions.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array; it can be of any data type.
SHIFT	Must be a scalar integer or an array with a rank that is one less than ARRAY, and shape ($d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n$), where ($d_1, d_2,..., d_n$) is the shape of ARRAY.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY. If DIM is omitted, it is assumed to be 1.

Results: The result is an array with the same type and kind parameters, and shape as ARRAY.

If ARRAY has rank one, element i of the result is ARRAY (1 + MODULO (i + SHIFT − 1, SIZE (ARRAY))). (The same shift is applied to each element.)

If ARRAY has rank greater than one, each section ($s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$) of the result is shifted as follows:

- By the value of SHIFT, if SHIFT is scalar

- According to the corresponding value in SHIFT(s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n), if SHIFT is an array

The value of SHIFT determines the amount and direction of the circular shift. A positive SHIFT value causes a shift to the left (in rows) or up (in columns). A negative SHIFT value causes a shift to the right (in rows) or down (in columns). A zero SHIFT value causes no shift.

Examples

V is the array (1, 2, 3, 4, 5, 6).

CSHIFT (V, SHIFT=2) shifts the elements in V circularly to the left by two positions, producing the value (3, 4, 5, 6, 1, 2). 1 and 2 are shifted off the beginning and inserted at the end.

CSHIFT (V, SHIFT= –2) shifts the elements in V circularly to the right by two positions, producing the value (5, 6, 1, 2, 3, 4). 5 and 6 are shifted off the end and inserted at the beginning.

M is the array $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$.

CSHIFT (M, SHIFT = 1, DIM = 2) produces the result $\begin{bmatrix}2 & 3 & 1\\5 & 6 & 4\\8 & 9 & 7\end{bmatrix}$.

Each element in rows 1, 2, and 3 is shifted to the left by two positions. The elements shifted off the beginning are inserted at the end.

CSHIFT (M, SHIFT = –1, DIM = 1) produces the result $\begin{bmatrix}7 & 8 & 9\\1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$.

Each element in columns 1, 2, and 3 is shifted down by one position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, –1, 0/), DIM = 2) produces the result $\begin{bmatrix}2 & 3 & 1\\6 & 4 & 5\\7 & 8 & 9\end{bmatrix}$.

Each element in row 1 is shifted to the left by one position; each element in row 2 is shifted to the right by one position; no element in row 3 is shifted at all.

9.4.34. DATE (BUF)

Description: Returns the current date as set within the system.

Class: Subroutine

Arguments: BUF is a 9-byte variable, array, array element, or character substring.

The date is returned as a 9-byte ASCII character string taking the form dd-mmm-yy, where:

dd is the 2-digit date
mmm is the 3-letter month

yy is the last two digits of the year

If BUF is of numeric type and smaller than 9 bytes, data corruption can occur.

If BUF is of character type, its associated length is passed to the subroutine. If BUF is smaller than 9 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Warning: The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead (see Section 9.4.35).

Examples

Consider the following:

```
CHARACTER*1 DAY(9)
...
CALL DATE (DAY)
```

The length of the first array element in CHARACTER array DAY is passed to the DATE subroutine. The subroutine then truncates the date to fit into the one-character element, producing an incorrect result.

9.4.35. DATE_AND_TIME ([DATE] [,TIME] [,ZONE] [,VALUES])

Description:	Returns character data on the real-time clock and date in a form compatible with the representations defined in Standard ISO 8601:1988.
Class:	Subroutine
Arguments:	There are four optional arguments[1]:
DATE (opt)	Must be scalar and of type default character; its length must be at least 8 to contain the complete value. Its leftmost 8 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 *DD* is the day within the month
TIME (opt)	Must be scalar and of type default character; its length must be at least 10 to contain the complete value. Its leftmost 10 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 *ss.sss* is the seconds and milliseconds of the minute
ZONE (opt)	Must be scalar and of type default character; its length must be at least 5 to contain the complete value. Its leftmost 5 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)[2] in hours and parts of an hour expressed in minutes, respectively.
VALUES (opt)	Must be of type default integer and of rank one. Its size must be at least 8. The values returned in VALUES are as follows:

VALUES (1) is the 4-digit year.
VALUES (2) is the month of the year.
VALUES (3) is the day of the month.
VALUES (4) is the time difference with respect to Coordinated Universal Time (UTC) in minutes.
VALUES (5) is the hour of the day (range 0 to 23).[3]
VALUES (6) is the minutes of the hour (range 0 to 59).[3]
VALUES (7) is the seconds of the minute (range 0 to 59).[3]
VALUES (8) is the milliseconds of the second (range 0 to 999).[3]

[1]All are INTENT(OUT) arguments. (See Section 5.10).
[2]UTC (also known as Greenwich Mean Time) is defined by CCIR Recommendation 460–2.
[3]In local time.

Note: If time zone information is not available on the system, a blank is returned for the ZONE argument and –1 is returned for the differential element of the VALUES argument.

Examples

Consider the following example executed on 2000 March 28 at 11:04:14.5:

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 12) REAL_CLOCK (3)
CALL DATE_AND_TIME (REAL_CLOCK (1), REAL_CLOCK (2), &
                    REAL_CLOCK (3), DATE_TIME)
```

This assigns the value “20000328” to REAL_CLOCK (1), the value “110414.500” to REAL_CLOCK (2), and the value “–0500” to REAL_CLOCK (3). The following values are assigned to DATE_TIME: 2000, 3, 28, -300, 11, 4, 14, and 500.

9.4.36. DBLE (A)

Description: Converts a number to double-precision real type.

Class: Elemental function; Generic

Arguments: A must be of type integer, real, or complex.

Results: The result is of type double precision real (REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If A is of type double precision, the result is the value of the A with no conversion (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 value 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 value can contain.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
	INTEGER(2)	REAL(8)
	INTEGER(4)	REAL(8)
	INTEGER(8)	REAL(8)

Specific Name[1]	Argument Type	Result Type
DBLE[2]	REAL(4)	REAL(8)
	REAL(8)	REAL(8)
DBLEQ	REAL(16)	REAL(8)
	COMPLEX(4)	REAL(8)
	COMPLEX(8)	REAL(8)
	COMPLEX(16)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

[2]For compatibility with older versions of Fortran, DBLE can also be specified as a specific function.

Examples

DBLE (4) has the value 4.0.

DBLE ((3.4, 2.0)) has the value 3.4.

9.4.37. DCMPLX (X [,Y])

Description: Converts the argument to double complex type. This function must not be passed as an actual argument.

Class: Elemental function; Generic

Arguments:

X	Must be of type integer, real, or complex.
Y (opt)	Must be of type integer or real. It must not be present if X is of type complex.

Results: The result is of type double complex (COMPLEX(8) or COMPLEX*16).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If Y is not specified and X is complex, the result value is CMPLX (REAL(X), AIMAG(X)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

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 (–3.0, 0.0).

DCMPLX (4.1, 2.3) has the value (4.1, 2.3).

9.4.38. DFLOAT (A)

Description: Converts an integer to double-precision type.

Class: Elemental function; Generic

Arguments: A must be of type integer.

Results: The result is of type double-precision real (REAL(8) or REAL*8).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
DFLOTI	INTEGER(2)	REAL(8)
DFLOTJ	INTEGER(4)	REAL(8)
DFLOTK	INTEGER(8)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

Examples

DFLOAT (–4) has the value –4.0.

9.4.39. DIGITS (X)

Description: Returns the number of significant binary digits for numbers of the same type and kind parameters as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type integer or real; it can be scalar or array valued.

Results: The result is a scalar of type default integer.

The result has the value *q* if X is of type integer; it has the value *p* if X is of type real. Integer parameter *q* is defined in Section D.1; real parameter *p* is defined in Section D.2.

Examples

If X is of type REAL(4), DIGITS (X) has the value 24.

9.4.40. DIM (X, Y)

Description: Returns the difference between two numbers (if the difference is positive).

Class: Elemental function; Generic

Arguments: X — Must be of type integer or real.

Y — Must have the same type and kind parameters as X.

Results: The result type is the same as X. The value of the result is X – Y if X is greater than Y; otherwise, the value of the result is zero.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIDIM	INTEGER(2)	INTEGER(2)
IDIM[1]	INTEGER(4)	INTEGER(4)
KIDIM	INTEGER(8)	INTEGER(8)
DIM	REAL(4)	REAL(4)
DDIM	REAL(8)	REAL(8)
QDIM	REAL(16)	REAL(16)

[1]Or JIDIM.

Examples

DIM (6, 2) has the value 4.

DIM (–4.0, 3.0) has the value 0.0.

9.4.41. DOT_PRODUCT (VECTOR_A, VECTOR_B)

Description:	Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).	
Class:	Transformational function; Generic	
Arguments:	VECTOR_A	Must be a rank-one array of numeric (integer, real, or complex) or logical type.
	VECTOR_B	Must be a rank-one array of numeric type if VECTOR_A is of numeric type, or of logical type if VECTOR_A is of logical type. It must be the same size as VECTOR_A.
Results:	The result is a scalar whose type depends on the types of VECTOR_A and VECTOR_B. If VECTOR_A is of type integer or real, the result value is SUM (VECTOR_A*VECTOR_B). If VECTOR_A is of type complex, the result value is SUM (CONJG (VECTOR_A)*VECTOR_B). If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If either rank-one array has size zero, the result is zero if the array is of numeric type, and false if the array is of logical type. (For more information on expressions, see Section 4.1).	

Examples

DOT_PRODUCT ((/1, 2, 3/), (/3, 4, 5/)) has the value 26 (calculated as follows: ((1 × 3) + (2 × 4) + (3 × 5)) = 26).

DOT_PRODUCT ((/ (1.0, 2.0), (2.0, 3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /)) has the value (17.0, 4.0).

DOT_PRODUCT ((/ .TRUE., .FALSE. /), (/ .FALSE., .TRUE. /)) has the value false.

9.4.42. DPROD (X, Y)

Description:	Produces a higher precision product. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.	
Class:	Elemental function; Specific	
Arguments:	X	Must be of type REAL(4) or REAL(8).
	Y	Must be the same type and kind parameter as X.
Results:	If X and Y are of type REAL(4), the result is of type double-precision real. If X and Y are of type REAL(8), the result is of type REAL(16). The result value is equal to X*Y.	

Examples

DPROD (2.0, –4.0) has the value –8.00D0.

DPROD (5.0D0, 3.0D0) has the value 15.00Q0.

```
  REAL(4) e
  REAL(8) d
  e = 123456.7
  d = 123456.7D0
  ! DPROD (e,e) returns 15241557546.4944

  ! DPROD (d,d) returns 15241556774.889999281387426890432 8
```

9.4.43. DREAL (A)

Description:	Converts the real part of a double-complex argument to double-precision type. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.
Class:	Elemental function; Specific
Arguments:	A must be of type double complex (COMPLEX(8) or COMPLEX*16).
Results:	The result is of type double-precision real (REAL(8) or REAL*8).

Examples

DREAL ((2.0d0, 3.0d0)) has the value 2.0d0.

9.4.44. EOF (A)

Description:	Checks whether a file is at or beyond the end-of-file record. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.
Class:	Inquiry function; Specific
Arguments:	A must be of type integer. It represents a unit specifier corresponding to an open file. It cannot be zero unless you have reconnected unit zero to a unit other than the screen or keyboard.
Results:	The result is of type logical. The value of the result is .TRUE. if the file connected to A is at or beyond the end-of-file record; otherwise, .FALSE..

Examples

Consider the following:

```
!  Creates a file of random numbers, reads them back
      REAL x, total
      INTEGER count
      OPEN (1, FILE = 'TEST.DAT')
      DO I = 1, 20
        CALL RANDOM_NUMBER(x)
        WRITE (1, '(F6.3)') x * 100.0
      END DO
      CLOSE(1)
      OPEN (1, FILE = 'TEST.DAT')
      DO WHILE (.NOT. EOF(1))
```

```
        count = count + 1
        READ (1, *) value
        total = total + value
      END DO
100   IF ( count .GT. 0) THEN
        WRITE (*,*) 'Average is: ', total / count
      ELSE
        WRITE (*,*) 'Input file is empty '
      END IF
 STOP
      END
```

9.4.45. EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])

Description: Performs an *end-off* shift on a rank-one array, or performs end-off shifts on all the complete rank-one sections along a given dimension of an array of rank two or greater.

Elements are shifted off at one end of a section and copies of a boundary value are filled in at the other end. Different sections can have different boundary values and can be shifted by different amounts and in different directions.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array (of any data type).
SHIFT	Must be a scalar integer or an array with a rank that is one less than ARRAY, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of ARRAY.
BOUNDARY (opt)	Must have the same type and kind parameters as ARRAY. It must be a scalar or an array with a rank that is one less than ARRAY, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n). If BOUNDARY is not specified, it is assumed to have the following default values (depending on the data type of ARRAY):

ARRAY Type	**BOUNDARY Value**
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character (*len*)	*len* blanks

DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY. If DIM is omitted, it is assumed to be 1.

Results: The result is an array with the same type and kind parameters, and shape as ARRAY.

If ARRAY has rank one, the same shift is applied to each element. If an element is shifted off one end of the array, the BOUNDARY value is placed at the other end the array.

If ARRAY has rank greater than one, each section ($s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$) of the result is shifted as follows:

- By the value of SHIFT, if SHIFT is scalar
- According to the corresponding value in SHIFT($s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n$), if SHIFT is an array

If an element is shifted off one end of a section, the BOUNDARY value is placed at the other end of the section.

The value of SHIFT determines the amount and direction of the end-off shift. A positive SHIFT value causes a shift to the left (in rows) or up (in columns). A negative SHIFT value causes a shift to the right (in rows) or down (in columns).

Examples

V is the array (1, 2, 3, 4, 5, 6).

EOSHIFT (V, SHIFT=2) shifts the elements in V to the left by two positions, producing the value (3, 4, 5, 6, 0, 0). 1 and 2 are shifted off the beginning and two elements with the default BOUNDARY value are placed at the end.

EOSHIFT (V, SHIFT= –3, BOUNDARY= 99) shifts the elements in V to the *right* by 3 positions, producing the value (99, 99, 99, 1, 2, 3). 4, 5, and 6 are shifted off the end and three elements with BOUNDARY value 99 are placed at the beginning.

M is the array $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$.

EOSHIFT (M, SHIFT = 1, BOUNDARY = '*', DIM = 2) produces the result $\begin{bmatrix}2 & 3 & *\\5 & 6 & *\\8 & 9 & *\end{bmatrix}$.

Each element in rows 1, 2, and 3 is shifted to the left by one position. This causes the first element in each row to be shifted off the beginning, and the BOUNDARY value to be placed at the end.

EOSHIFT (M, SHIFT = –1, DIM = 1) produces the result $\begin{bmatrix}0 & 0 & 0\\1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$.

Each element in columns 1, 2, and 3 is shifted *down* by 1 position. This causes the last element in each column to be shifted off the end and the BOUNDARY value to be placed at the beginning.

EOSHIFT (M, SHIFT = (/1, –1, 0/), BOUNDARY = (/ '*', '?', '/ ' /), DIM = 2) produces the result $\begin{bmatrix}2 & 3 & *\\? & 4 & 5\\7 & 8 & 9\end{bmatrix}$.

Each element in row 1 is shifted to the left by one position, causing the first element to be shifted off the beginning and the BOUNDARY value * to be placed at the end. Each element in row 2 is shifted to the *right* by 1 position, causing the last element to be shifted off the end and the BOUNDARY value ? to be placed at the beginning. No element in row 3 is shifted at all, so the specified BOUNDARY value is not used.

9.4.46. EPSILON (X)

Description:	Returns the difference (for scalars of the same type and kind parameters) between 1.0 and the next larger model number. EPSILON is a guide to the precision with which values near unity can be represented. EPSILON(1.0) is about 1.19E-7, EPSILON(1.0_8) is about 2.22E-16, and EPSILON(1.0_16) is about 1.93E-34.
Class:	Inquiry function; Generic
Arguments:	X must be of type real; it can be scalar or array valued.
Results:	The result is a scalar of the same type and kind parameters as X. The result has the value b^{1-p}. Parameters *b* and *p* are defined in Section D.2.

Examples

If X is of type REAL(4), EPSILON (X) has the value 2^{-23}.

9.4.47. ERRSNS ([IO_ERR] [,SYS_ERR] [,STAT] [,UNIT] [,COND])

Description:	Returns information about the most recently detected I/O system error condition.
Class:	Subroutine
Arguments:	There are five optional arguments:
IO_ERR (opt)	Is an integer variable or array element that stores the most recent VSI Fortran RTL error number that occurred during program execution. (For a listing of error numbers, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].) A zero indicates no error has occurred since the last call to ERRSNS or since the start of program execution.
SYS_ERR (opt)	Is an integer variable or array element that stores the most recent system error number associated with IO_ERR. This code is an RMS STS value.
STAT (opt)	Is an integer variable or array element that stores a status value that occurred during program execution. This value is an RMS STV value.
UNIT (opt)	Is an integer variable or array element that stores the logical unit number, if the last error was an I/O error.
COND (opt)	Is an integer variable or array element that stores the actual processor value. This value is always zero. If you specify INTEGER(2) arguments, only the low-order 16 bits of information are returned or adjacent data can be overwritten. Because of this, it is best to use INTEGER(4) arguments. The saved error information is set to zero after each call to ERRSNS.

Examples

Any of the arguments can be omitted. For example, the following is valid:

```
CALL ERRSNS (SYS_ERR, STAT, , UNIT)
```

9.4.48. EXIT ([STATUS])

Description: Terminates program execution, closes all files, and returns control to the operating system.

Class: Subroutine

Arguments: STATUS is an optional integer argument you can use to specify the image exit-status value.

Examples

CALL EXIT (100)

9.4.49. EXP (X)

Description: Computes an exponential value.

Class: Elemental function; Generic

Arguments: X must be of type real or complex.

Results: The result type is the same as X. The value of the result is e^{x}. If X is of type complex, its imaginary part is regarded as a value in radians.

Specific Name	Argument Type	Result Type
EXP	REAL(4)	REAL(4)
DEXP	REAL(8)	REAL(8)
QEXP	REAL(16)	REAL(16)
CEXP[1]	COMPLEX(4)	COMPLEX(4)
CDEXP[2]	COMPLEX(8)	COMPLEX(8)
CQEXP	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CEXP.

[2]This function can also be specified as ZEXP.

Examples

EXP (2.0) has the value 7.389056.

EXP (1.3) has the value 3.669297.

9.4.50. EXPONENT (X)

Description: Returns the exponent part of the argument when represented as a model number.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result is of type default integer. If X is not equal to zero, the result value is the exponent part of X. The exponent must be within default integer range; otherwise, the result is undefined.

If X is zero, the exponent of X is zero. For more information on the exponent part (*e*) in the real model, see Section D.2.

Examples

EXPONENT (2.0) has the value 2.

If 4.1 is a REAL(4) value, EXPONENT (4.1) has the value 3.

9.4.51. FLOOR (A [,KIND])

Description:	Returns the greatest integer less than or equal to its argument.	
Class:	Elemental function; Generic	
Arguments:	A	Must be of type real.
	KIND (opt)	Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.
Results:	The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined. The value of the result is equal to the greatest integer less than or equal to A.	

Examples

FLOOR (4.8) has the value 4.

FLOOR (–5.6) has the value –6.

9.4.52. FP_CLASS (X)

Description: Returns the class of an IEEE real (S_floating, T_floating, or

X_floating

) argument. The compiler option specifying IEEE floating format must be set.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result is of type default integer. The return value is one of the following:

```
Class of Argument
Return Value

Signaling NaN FOR_K_FP_SNAN

Quiet NaN FOR_K_FP_QNAN

Positive Infinity FOR_K_FP_POS_INF

Negative Infinity FOR_K_FP_NEG_INF

Positive Normalized Number FOR_K_FP_POS_NORM

Negative Normalized Number FOR_K_FP_NEG_NORM
```

```
Positive Denormalized Number FOR_K_FP_POS_DENORM

Negative Denormalized Number FOR_K_FP_NEG_DENORM

Positive Zero FOR_K_FP_POS_ZERO

Negative Zero FOR_K_FP_NEG_ZERO
```

The preceding return values are defined in module FORSYSDEF. For information on the location of this file, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Examples

FP_CLASS (4.0_8) has the value 4 (FOR_K_FP_POS_NORM, a normal positive number).

9.4.53. FRACTION (X)

Description: Returns the fractional part of the model representation of the argument value.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X. The result has the value $X \times b^{-e}$. Parameters *b* and *e* are defined in Section D.2. If X has the value zero, the result has the value zero.

Examples

If 3.0 is a REAL(4) value, FRACTION (3.0) has the value 0.75.

9.4.54. FREE (A)

Description: Frees a block of memory that is currently allocated.

Class: Subroutine

Arguments: A must be of type INTEGER(8). This value is the starting address of the memory to be freed, previously allocated by MALLOC (see Section 9.4.91).

If the freed address was not previously allocated by MALLOC, or if an address is freed more than once, results are unpredictable.

Examples

Consider the following:

```
INTEGER(8) ADDR, SIZE
SIZE = 1024                     ! Size in bytes
ADDR = MALLOC(SIZE)             ! Allocate the memory
CALL FREE(ADDR)                 ! Free it
END
```

9.4.55. HUGE (X)

Description: Returns the largest number in the model representing the same type and kind parameters as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type integer or real; it can be scalar or array valued.

Results: The result is a scalar of the same type and kind parameters as X. If X is of type integer, the result has the value $r^q - 1$. If X is of type real, the result has the value $(1 - b^{-p}) b^{e_{max}}$.

Integer parameters *r* and *q* are defined in Section D.1; real parameters *b*, *p*, and e_{max} are defined in Section D.2.

Examples

If X is of type REAL(4), HUGE (X) has the value $(1 - 2^{-24}) \times 2^{128}$.

9.4.56. IACHAR (C)

Description: Returns the position of a character in the ASCII character set, even if the processor's default character set is different. In VSI Fortran, IACHAR is equivalent to the ICHAR function.

Class: Elemental function; Generic

Arguments: C must be of type character of length 1.

Results: The result is of type default integer. If C is in the ASCII collating sequence, the result is the position of C in that sequence and satisfies the inequality (0 <= IACHAR(C) <= 127).

The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is true, IACHAR(C) .LE. IACHAR(D) is also true.

Examples

IACHAR ('Y') has the value 89.

IACHAR ('%') has the value 37.

9.4.57. IAND (I, J)

Description: Performs a logical AND on corresponding bits.[1]

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
J	Must be of type integer with the same kind parameter as I.

Results: The result type is the same as I. The result value is derived by combining I and J bit-by-bit according to the following truth table:

```
I J IAND (I, J)

1 1 1

1 0 0

0 1 0

0 0 0
```

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

[1]This function can also be specified as AND.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIAND	INTEGER(2)	INTEGER(2)
JIAND	INTEGER(4)	INTEGER(4)
KIAND	INTEGER(8)	INTEGER(8)

Examples

IAND (2, 3) has the value 2.

IAND (4, 6) has the value 4.

9.4.58. IARGCOUNT ( )

Description: Returns the count of actual arguments passed to the current routine.

Class: Inquiry function; Specific

Arguments: None.

Results: The result is of type default integer. Functions with a type of CHARACTER, COMPLEX(8), REAL(16), and COMPLEX(16) have an extra argument added that is used to return the function value.

Formal (dummy) arguments that can be omitted must be declared VOLATILE. For more information, see Section 5.19.

Formal arguments of type CHARACTER cannot be omitted. Formal arguments that are adjustable arrays (see Section 5.1.4.1) cannot be omitted.

The standard way to pass and detect omitted arguments is to use the Fortran 95 features of OPTIONAL arguments and the PRESENT intrinsic function. Note that a declaration must be visible within the calling routine.

Examples

Consider the following:

```
CALL SUB (A,B)
...
SUBROUTINE SUB (X,Y,Z)
VOLATILE Z
TYPE *, IARGCOUNT()       ! Displays the value 2
```

For more information, including an example, see Section 8.8.1.2.

9.4.59. IARGPTR ( )

Description: Returns a pointer to the actual argument list for the current routine.

Class: Inquiry function; Specific

Arguments: None.

Results: The result is of type INTEGER(8). The actual argument list is an array of values of the same type.

The first element in the array contains the argument count; subsequent elements contain the INTEGER(8) address of the actual arguments.

Formal (dummy) arguments that can be omitted must be declared VOLATILE. For more information, see Section 5.19.

Examples

Consider the following:

```
WRITE (*,'(" Address of argument list is ",Z16.8)') IARGPTR()
```

9.4.60. IBCHNG (I, POS)

Description: Reverses the value of a specified bit in an integer.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer. This argument contains the bit to be reversed.
POS	Must be of type integer. This argument is the position of the bit to be changed. The rightmost (least significant) bit of I is in position 0.

Results: The result type is the same as I. The result is equal to I with the bit in position POS reversed.

For more information on bit functions, see Section 9.3.3.

Examples

Consider the following:

```
INTEGER J, K
J = IBCHNG(10, 2)          ! returns 14 = 1110
K = IBCHNG(10, 1)          ! returns  8 = 1000
```

9.4.61. IBCLR (I, POS)

Description: Clears one bit to zero.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
POS	Must be of type integer. It must not be negative and it must be less than BIT_SIZE (I). The rightmost (least significant) bit of I is in position 0.

Results: The result type is the same as I. The result has the value of the sequence of bits of I, except that bit POS of I is set to zero. The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

For more information on bit functions, see Section 9.3.3.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBCLR	INTEGER(2)	INTEGER(2)
JIBCLR	INTEGER(4)	INTEGER(4)
KIBCLR	INTEGER(8)	INTEGER(8)

Examples

IBCLR (18, 1) has the value 16.

If V has the value (1, 2, 3, 4), the value of IBCLR (POS = V, I = 15) is (13, 11, 7, 15).

9.4.62. IBITS (I, POS, LEN)

Description: Extracts a sequence of bits (a bit field).

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
POS	Must be of type integer. It must not be negative and POS + LEN must be less than or equal to BIT_SIZE (I) . The rightmost (least significant) bit of I is in position 0.
LEN	Must be of type integer. It must not be negative.

Results: The result type is the same as I. The result has the value of the sequence of LEN bits in I, beginning at POS right-adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

For more information on bit functions, see Section 9.3.3.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBITS	INTEGER(2)	INTEGER(2)
JIBITS	INTEGER(4)	INTEGER(4)
KIBITS	INTEGER(8)	INTEGER(8)

Examples

IBITS (12, 1, 4) has the value 6.

IBITS (10, 1, 7) has the value 5.

9.4.63. IBSET (I, POS)

Description: Sets one bit to 1.

Class:	Elemental function; Generic	
Arguments:	I	Must be of type integer.
	POS	Must be of type integer. It must not be negative and it must be less than BIT_SIZE (I). The rightmost (least significant) bit of I is in position 0.
Results:	The result type is the same as I. The result has the value of the sequence of bits of I, except that bit POS of I is set to 1. The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3. For more information on bit functions, see Section 9.3.3.	

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBSET	INTEGER(2)	INTEGER(2)
JIBSET	INTEGER(4)	INTEGER(4)
KIBSET	INTEGER(8)	INTEGER(8)

Examples

IBSET (8, 1) has the value 10.

If V has the value (1, 2, 3, 4), the value of IBSET (POS = V, I = 2) is (2, 6, 10, 18).

9.4.64. ICHAR (C)

Description:	Returns the position of a character in the processor's character set.
Class:	Elemental function; Generic
Arguments:	C must be of type character of length 1.
Results:	The result is of type default integer. The result value is the position of C in the processor's character set. C is in the range zero to n − 1, where *n* is the number of characters in the character set. For any characters C and D (capable of representation in the processor), C .LE. D is true only if ICHAR(C) .LE. ICHAR(D) is true, and C .EQ. D is true only if ICHAR(C) .EQ. ICHAR(D) is true.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(2)
ICHAR[1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

[1]This specific function cannot be passed as an actual argument.

Examples

ICHAR ('W') has the value 87.

ICHAR ('#') has the value 35.

9.4.65. IDATE (I, J, K)

Description: Returns three integer values representing the current month, day, and year.

Class: Subroutine

Arguments: I is the current month.

J is the current day.

K is the current year.

Warning: The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead (see Section 9.4.35).

Note: If time-zone information is not available on the system, a blank is returned for the ZONE argument and –1 is returned for the differential element of the VALUES argument.

Examples

If the current date is September 16, 1996, the values of the integer variables upon return are: I = 9, J = 16, and K = 96.

9.4.66. IEOR (I, J)

Description: Performs an exclusive OR on corresponding bits.[1]

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
J	Must be of type integer with the same kind parameter as I.

Results: The result type is the same as I. The result value is derived by combining I and J bit-by-bit according to the following truth table:

```
I J IEOR (I, J)

1 1 0

1 0 1

0 1 1

0 0 0
```

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

[1]This function can also be specified as XOR.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIEOR	INTEGER(2)	INTEGER(2)
JIEOR	INTEGER(4)	INTEGER(4)
KIEOR	INTEGER(8)	INTEGER(8)

Examples

IEOR (1, 4) has the value 5.

IEOR (3, 10) has the value 9.

9.4.67. ILEN (I)

Description:	Returns the length (in bits) of the two's complement representation of an integer.
Class:	Elemental function; Generic
Arguments:	I must be of type integer.
Results:	The result type is the same as I. The result value is (LOG $_2$(I + 1)) if I is not negative; otherwise, the result value is (LOG $_2$(−I)).

Examples

ILEN (4) has the value 3.

ILEN (–4) has the value 2.

9.4.68. INDEX (STRING, SUBSTRING [,BACK] [,KIND])

Description:	Returns the starting position of a substring within a string.	
Class:	Elemental function; Generic	
Arguments:	STRING	Must be of type character.
	SUBSTRING	Must be of type character.
	BACK (opt)	Must be of type logical.
	KIND (opt)	Must be a scalar integer initialization expression.
Results:	The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined. If BACK is absent or false, the value returned is the minimum value of I such that STRING (I : I + LEN (SUBSTRING) − 1) = SUBSTRING (or zero if there is no such value). If LEN (STRING) < LEN (SUBSTRING), zero is returned. If LEN (SUBSTRING) = zero, 1 is returned. If BACK is true, the value returned is the maximum value of I such that STRING (I : I + LEN (SUBSTRING) − 1) = SUBSTRING (or zero if there is no such value). If LEN(STRING) < LEN (SUBSTRING), zero is returned. If LEN (SUBSTRING) = zero, LEN (STRING) + 1 is returned.	

Specific Name	Argument Type	Result Type
INDEX	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

Examples

INDEX ('FORTRAN', 'O', BACK = .TRUE.) has the value 2.

INDEX ('XXXX', "Δ", BACK = .TRUE.) has the value 0.

INDEX ('XXXX', " ", BACK = .TRUE.) has the value 5.

9.4.69. INT (A [,KIND])

Description: Converts a value to integer type.

Class: Elemental function; Generic

Arguments:

	A	Must be of type integer, real, or complex.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

The result value depends on the type and absolute value of A:

- If A is of type integer, INT (A) = A.
- If A is of type real and |A | < 1, INT (A) has the value zero.

  If A is of type real and |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.

  If A is of type complex, INT (A) = A is the value obtained by applying the preceding rules (for a real argument) to the real part of A.

The setting of compiler options specifying integer size can affect INT, IDINT, and IQINT.

The setting of compiler options specifying integer size or real size can affect IFIX.

Specific Name[1]	**Argument Type**	**Result Type**
	INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)
	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8)	INTEGER(8)
IIFIX[2]	REAL(4)	INTEGER(2)
IINT	REAL(4)	INTEGER(2)
IFIX[3]	REAL(4)	INTEGER(4)
JFIX	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
INT[4]	REAL(4)	INTEGER(4)
KIFIX	REAL(4)	INTEGER(8)
KINT	REAL(4)	INTEGER(8)
IIDINT	REAL(8)	INTEGER(2)

Specific Name[1]	Argument Type	Result Type
IDINT[5]	REAL(8)	INTEGER(4)
KIDINT	REAL(8)	INTEGER(8)
IIQINT	REAL(16)	INTEGER(2)
IQINT[6]	REAL(16)	INTEGER(4)
KIQINT	REAL(16)	INTEGER(8)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)
INT1	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(1)
INT2	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
INT4	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
INT8	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)

[1]These specific functions cannot be passed as actual arguments.

[2]This function can also be specified as HFIX.

[3]For compatibility with older versions of Fortran, IFIX can also be specified as a generic function.

[4]Or JINT.

[5]Or JIDINT. For compatibility with older versions of Fortran, IDINT can also be specified as a generic function.

[6]Or JIQINT. For compatibility with older versions of Fortran, IQINT can also be specified as a generic function.

Examples

INT (–4.2) has the value –4.

INT (7.8) has the value 7.

9.4.70. INT_PTR_KIND( )

Description: Returns the INTEGER KIND that will hold an address. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Inquiry function; Specific

Arguments: None.

Results: The result is of type default integer. The result is a scalar with the value equal to the value of the kind parameter of the integer data type that can represent an address on the host platform.

The value is 8.

Examples

Consider the following:

```
REAL A(100)
POINTER (P, A)
INTEGER (KIND=INT_PTR_KIND()) SAVE_P
P = MALLOC (400)
SAVE_P = P
```

9.4.71. IOR (I, J)

Description: Performs an inclusive OR on corresponding bits.[1]

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
J	Must be of type integer with the same kind parameter as I.

Results: The result type is the same as I. The result value is derived by combining I and J bit-by-bit according to the following truth table:

```
I J IOR (I, J)

1 1 1

1 0 1

0 1 1

0 0 0
```

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

[1]This function can also be specified as OR.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIOR	INTEGER(2)	INTEGER(2)
JIOR	INTEGER(4)	INTEGER(4)
KIOR	INTEGER(8)	INTEGER(8)

Examples

IOR (1, 4) has the value 5.

IOR (1, 2) has the value 3.

9.4.72. ISHA (I, SHIFT)

Description: Arithmetically shifts an integer left or right by a specified number of bits.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer. This argument is the value to be shifted.
SHIFT	Must be of type integer. This argument is the direction and distance of shift. Positive shifts are left (toward the most significant bit); negative shifts are right (toward the least significant bit).

Results: The result type is the same as I. The result is equal to I shifted arithmetically by SHIFT bits.

If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right. If SHIFT is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. If the shift is to the left, zeros are shifted in on the right. If the shift is to the right, copies of the sign bit (0 for non-negative I; 1 for negative I) are shifted in on the left.

The kind of integer is important in arithmetic shifting because sign varies among integer representations (see the following example). If you want to shift a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Examples

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = -128             ! equal to  10000000
j = -32768           ! equal to  10000000 00000000
res1  = ISHA (i, -4) ! returns 11111000 = -8
res2  = ISHA (j, -4) ! returns 11111000 00000000 = -2048
```

9.4.73. ISHC (I, SHIFT)

Description: Rotates an integer left or right by specified number of bits. Bits shifted out one end are shifted in the other end. No bits are lost.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer. This argument is the value to be rotated.
SHIFT	Must be of type integer. This argument is the direction and distance of rotation. Positive rotations are left (toward the most significant bit); negative rotations are right (toward the least significant bit).

Results: The result type is the same as I. The result is equal to I circularly rotated by SHIFT bits.

If SHIFT is positive, I is rotated left SHIFT bits. If SHIFT is negative, I is rotated right SHIFT bits. Bits shifted out one end are shifted in the other. No bits are lost.

The kind of integer is important in circular shifting. With an INTEGER(4) argument, all 32 bits are shifted. If you want to rotate a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Examples

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10                  ! equal to  00001010
j = 10                  ! equal to  00000000 00001010
res1  = ISHC (i, -3)    ! returns 01000001 =  65
res2  = ISHC (j, -3)    ! returns 01000000 00000001 = 16385
```

9.4.74. ISHFT (I, SHIFT)

Description:	Performs a logical shift.	
Class:	Elemental function; Generic	
Arguments:	I	Must be of type integer.
	SHIFT	Must be of type integer. The absolute value for SHIFT must be less than or equal to BIT_SIZE (I) .

Results: The result type is the same as I. The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is positive[1], the shift is to the left; if SHIFT is negative[2], the shift is to the right. If SHIFT is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3. For more information on bit functions, see Section 9.3.3.

[1]ISHFT with a positive SHIFT can also be specified as LSHIFT.

[2]ISHFT with a negative SHIFT can also be specified as RSHIFT with |SHIFT |.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IISHFT	INTEGER(2)	INTEGER(2)
JISHFT	INTEGER(4)	INTEGER(4)
KISHFT	INTEGER(8)	INTEGER(8)

Examples

ISHFT (2, 1) has the value 4.

ISHFT (2, –1) has the value 1.

9.4.75. ISHFTC (I, SHIFT [,SIZE])

Description: Performs a circular shift of the rightmost bits.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer.
SHIFT	Must be of type integer. The absolute value for SHIFT must be less than or equal to SIZE.
SIZE (opt)	Must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE (I) . If SIZE is omitted, it is assumed to have the value of BIT_SIZE (I) .

Results: The result type is the same as I. The result value is obtained by circular shifting the SIZE rightmost 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. If SHIFT is zero, no shift is performed.

No bits are lost. Bits in I beyond the value specified by SIZE are unaffected.

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3. For more information on bit functions, see Section 9.3.3.

Specific Name	Argument Type	Result Type
IISHFTC	INTEGER(2)	INTEGER(2)
JISHFTC	INTEGER(4)	INTEGER(4)
KISHFTC	INTEGER(8)	INTEGER(8)

Examples

ISHFTC (4, 2, 4) has the value 1.

ISHFTC (3, 1, 3) has the value 6.

9.4.76. ISHL (I, SHIFT)

Description: Logically shifts an integer left or right by the specified bits. Zeros are shifted in from the opposite end.

Class: Elemental function; Generic

Arguments:

I	Must be of type integer. This argument is the value to be shifted.
SHIFT	Must be of type integer. This argument is the direction and distance of shift. If positive, I is shifted left (toward the most significant bit). If negative, I is shifted right (toward the least significant bit).

Results: The result type is the same as I. The result is equal to I logically shifted by SHIFT bits. Zeros are shifted in from the opposite end.

Unlike circular or arithmetic shifts, which can shift ones into the number being shifted, logical shifts shift in zeros only, regardless of the direction or size of the shift. The integer kind, however, still determines the end that bits are shifted out of, which can make a difference in the result (see the following example).

Examples

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10                ! equal to  00001010
j = 10                ! equal to  00000000 00001010
res1  = ISHL (i, 5)   ! returns 01000000 = 64
res2  = ISHL (j, 5)   ! returns 00000001 01000000 = 320
```

9.4.77. ISNAN (X)

Description: Tests whether IEEE real (S_floating and T_floating) numbers are Not-a-Number (NaN) values. The compiler option /FLOAT=IEEE_FLOAT must be set.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result is of type default logical. The result is .TRUE. if X is an IEEE NaN; otherwise, the result is .FALSE..

Examples

Consider the following:

```
LOGICAL A
DOUBLE PRECISION B
...
A = ISNAN(B)
```

A is assigned the value .TRUE. if B is an IEEE NaN; otherwise, the value assigned is .FALSE..

9.4.78. KIND (X)

Description: Returns the value of the kind type parameter of the argument. For more information on kind type parameters, see Section 3.2.

Class: Inquiry function; Generic

Arguments: X can be of any intrinsic type.

Results: The result is a scalar of type default integer. The result has a value equal to the kind type parameter value of X.

Examples

KIND (0.0) has the kind value of default real type.

KIND (12) has the kind value of default integer type.

9.4.79. LBOUND (ARRAY [,DIM] [,KIND])

Description: Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.

Class: Inquiry function; Generic

Arguments:

ARRAY	Must be an array (of any data type). It must not be an allocatable array that is not allocated, or a disassociated pointer.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If DIM is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of ARRAY. Each element in the result corresponds to a dimension of ARRAY.

If ARRAY is an array section or an array expression that is not a whole array or array structure component, each element of the result has the value 1.

If ARRAY is a whole array or array structure component, LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of ARRAY (if ARRAY(DIM) is nonzero). If ARRAY(DIM) has size zero, the corresponding element of the result has the value 1.

The setting of compiler options that specify integer size can affect the result of this function.

Examples

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

LBOUND (ARRAY_A) is (1, 5). LBOUND (ARRAY_A, DIM=2) is 5.

LBOUND (ARRAY_B) is (2, –3). LBOUND (ARRAY_B (5:8, :)) is (1,1) because the arguments are array sections.

9.4.80. LEADZ (I)

Description: Returns the number of leading zero bits in an integer.

Class: Elemental function; Generic

Arguments: I must be of type integer.

Results: The result type is the same as I. The result value is the number of leading zeros in the binary representation of the integer I.

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

Examples

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
  TYPE *, LEADZ(TWO**J)  ! Prints 64 down to 23 (leading zeros)
ENDDO
END
```

9.4.81. LEN (STRING [,KIND])

Description:	Returns the length of a character expression.	
Class:	Inquiry function; Generic	
Arguments:	STRING	Must be of type character; it can be scalar or array valued.
	KIND (opt)	Must be a scalar integer initialization expression.
Results:	The result is a scalar of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined. 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). The setting of compiler options that specify integer size can affect the result of this function.	

Specific Name	Argument Type	Result Type
LEN	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

Examples

Consider the following example:

```
CHARACTER (15) C (50)
CHARACTER (25) D
```

LEN (C) has the value 15, and LEN (D) has the value 25.

9.4.82. LEN_TRIM (STRING [,KIND])

Description:	Returns the length of the character argument without counting trailing blank characters.	
Class:	Elemental function; Generic	
Arguments:	STRING	Must be of type character.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is a scalar of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters remaining after any trailing blanks in STRING are removed. If the argument contains only blank characters, the result is zero.

Examples

LEN_TRIM ('ΔΔΔCΔΔDΔΔΔ') has the value 7.

LEN_TRIM ('ΔΔΔΔΔ') has the value 0.

9.4.83. LGE (STRING_A, STRING_B)

Description: Determines if a string is lexically greater than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In VSI Fortran, LGE is equivalent to the >= operator.

Class: Elemental function; Generic

Arguments:

STRING_A	Must be of type character.
STRING_B	Must be of type character.

Results: The result is of type default logical. 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.

The result is true if the strings are equal, both strings are of zero length, or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LGE[1]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

Examples

LGE ('ONE', 'SIX') has the value false.

LGE ('TWO', 'THREE') has the value true.

9.4.84. LGT (STRING_A, STRING_B)

Description: Determines whether a string is lexically greater than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In VSI Fortran, LGT is equivalent to the > operator.

Class: Elemental function; Generic

Arguments:

STRING_A	Must be of type character.
STRING_B	Must be of type character.

Results: The result is of type default logical. 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.

The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LGT[1]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

Examples

LGT ('TWO', 'THREE') has the value true.

LGT ('ONE', 'FOUR') has the value true.

9.4.85. LLE (STRING_A, STRING_B)

Description: Determines whether a string is lexically less than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In VSI Fortran, LLE is equivalent to the <= operator.

Class: Elemental function; Generic

Arguments:

STRING_A	Must be of type character.
STRING_B	Must be of type character.

Results: The result is of type default logical. 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.

The result is true if the strings are equal, both strings are of zero length, or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LLE[1]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

Examples

LLE ('TWO', 'THREE') has the value false.

LLE ('ONE', 'FOUR') has the value false.

9.4.86. LLT (STRING_A, STRING_B)

Description: Determines whether a string is lexically less than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In VSI Fortran, LLT is equivalent to the < operator.

Class: Elemental function; Generic

Arguments:

STRING_A	Must be of type character.
STRING_B	Must be of type character.

Results: The result is of type default logical. 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.

The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LLT[1]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

Examples

LLT ('ONE', 'SIX') has the value true.

LLT ('ONE', 'FOUR') has the value false.

9.4.87. LOC (X)

Description: Returns the internal address of a storage item.[1]

Class: Inquiry function; Generic

Arguments: X is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of an internal procedure or statement function. If it is a pointer, it must be defined and associated with a target.

Results: The result is of type INTEGER(8). The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

In the case of global symbolic constants, LOC returns the value of the constant rather than an address.

This function serves the same purpose as the %LOC built-in function.

[1]This specific function cannot be passed as an actual argument.

9.4.88. LOG (X)

Description: Returns the natural logarithm of the argument.

Class: Elemental function; Generic

Arguments: X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Results: The result type is the same as X. The result value is approximately equal to $\log_e X$.

If the arguments are complex, the result is the principal value of imaginary part ω in the range $-\pi < \omega <= \pi$. The imaginary part of the result is π if the real part of the argument is less than zero and the imaginary part of the argument is zero.

Specific Name	Argument Type	Result Type
ALOG [1]	REAL(4)	REAL(4)
DLOG	REAL(8)	REAL(8)
QLOG	REAL(16)	REAL(16)
CLOG[1]	COMPLEX(4)	COMPLEX(4)

Specific Name	Argument Type	Result Type
CDLOG[2]	COMPLEX(8)	COMPLEX(8)
CQLOG	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect ALOG and CLOG.

[2]This function can also be specified as ZLOG.

Examples

LOG (8.0) has the value 2.079442.

LOG (25.0) has the value 3.218876.

9.4.89. LOG10 (X)

Description: Returns the common logarithm of the argument.

Class: Elemental function; Generic

Arguments: X must be of type real. The value of X must be greater than zero.

Results: The result type is the same as X. The result has a value equal to $\log_{10}X$.

Specific Name	Argument Type	Result Type
ALOG10[1]	REAL(4)	REAL(4)
DLOG10	REAL(8)	REAL(8)
QLOG10	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect ALOG10.

Examples

LOG10 (8.0) has the value 0.9030900.

LOG10 (15.0) has the value 1.176091.

9.4.90. LOGICAL (L [,KIND])

Description: Converts the logical value of the argument to a logical value with different kind parameters.

Class: Elemental function; Generic

Arguments:

L	Must be of type logical.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type logical. If KIND is present, the kind parameter is that specified by KIND; otherwise, the kind parameter is that of default logical. The result value is that of L.

The setting of compiler options specifying integer size can affect this function.

Examples

LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical regardless of the kind parameter of logical variable L.

LOGICAL (.FALSE., 2) has the value false, with the kind parameter of INTEGER(KIND=2).

9.4.91. MALLOC (I)

Description: Allocates a block of memory. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Elemental function; Specific

Arguments: I must be of type integer. This value is the size (in bytes) of memory to be allocated.

Results: The result is of type INTEGER(8).

If the argument is INTEGER(8), a 64-bit (P2) space is allocated.

The result is the starting address of the allocated memory. The memory allocated can be freed by using the FREE intrinsic function (see Section 9.4.54).

Examples

Consider the following:

```
INTEGER(8) ADDR, SIZE
SIZE = 1024                 ! Size in bytes
ADDR = MALLOC(SIZE)         ! Allocate the memory
CALL FREE(ADDR)             ! Free it
END
```

9.4.92. MATMUL (MATRIX_A, MATRIX_B)

Description: Performs matrix multiplication of numeric or logical matrices.

Class: Transformational function; Generic

Arguments:

MATRIX_A	Must be an array of rank one or two. It must be of numeric (integer, real, or complex) or logical type.
MATRIX_B	Must be an array of rank one or two. It must be of numeric type if MATRIX_A is of numeric type or logical type if MATRIX_A is logical type. At least one argument must be of rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Results: The result is an array whose type depends on the data type of the arguments, according to the rules shown in Table 4.2. The rank and shape of the result depends on the rank and shapes of the arguments, as follows:

- If MATRIX_A has shape (n, m) and MATRIX_B has shape (m, k), the result is a rank-two array with shape (n, k).
- If MATRIX_A has shape (m) and MATRIX_B has shape (m, k), the result is a rank-one array with shape (k).
- If MATRIX_A has shape (n, m) and MATRIX_B has shape (m), the result is a rank-one array with shape (n).

If the arguments are of numeric type, element (i, j) of the result has the value SUM ((row i of MATRIX_A) * (column j of MATRIX_B)). If the arguments are of logical

type, element (i, j) of the result has the value ANY ((row i of MATRIX_A) .AND. (column j of MATRIX_B)).

Examples

A is matrix $\begin{bmatrix}2 & 3 & 4\\3 & 4 & 5\end{bmatrix}$, B is matrix $\begin{bmatrix}2 & 3\\3 & 4\\4 & 5\end{bmatrix}$, X is vector (1, 2), and Y is vector (1, 2, 3).

The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix}29 & 38\\38 & 50\end{bmatrix}$.

The result of MATMUL (X, A) is the vector-matrix product XA with the value (8, 11, 14).

The result of MATMUL (A, Y) is the matrix-vector product AY with the value (20, 26).

9.4.93. MAX (A1, A2 [,A3,...])

Description: Returns the maximum value of the arguments.

Class: Elemental function; Generic

Arguments: A1, A2, and A3 (opt) must all have the same type (integer or real) and kind parameters.

Results: For MAX0, AMAX1, DMAX1, QMAX1, IMAX0, JMAX0, and KMAX0, the result type is the same as the arguments. For MAX1, IMAX1, JMAX1, and KMAX1, the result is of type integer. For AMAX0, AIMAX0, AJMAX0, and AKMAX0, the result is of type real. The value of the result is that of the largest argument.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
	INTEGER(1)	REAL(4)
IMAX0	INTEGER(2)	INTEGER(2)
AIMAX0	INTEGER(2)	REAL(4)
MAX0[2]	INTEGER(4)	INTEGER(4)
AMAX0[4]	INTEGER(4)	REAL(4)
KMAX0	INTEGER(8)	INTEGER(8)
AKMAX0	INTEGER(8)	REAL(4)
IMAX1	REAL(4)	INTEGER(2)
MAX1[6]	REAL(4)	INTEGER(4)
KMAX1	REAL(4)	INTEGER(8)
AMAX1[7]	REAL(4)	REAL(4)
DMAX1	REAL(8)	REAL(8)
QMAX1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMAX0.

[4]In Fortran 95/90, AMAX0 and MAX1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[6]The setting of compiler options specifying integer size can affect MAX1.

[7]The setting of compiler options specifying real size can affect AMAX1.

Examples

MAX (2.0, –8.0, 6.0) has the value 6.0.

MAX (14, 32, –50) has the value 32.

9.4.94. MAXEXPONENT (X)

Description: Returns the maximum exponent in the model representing the same type and kind parameters as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type real; it can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value e_{max}, as defined in Section D.2.

Examples

If X is of type REAL(4), MAXEXPONENT (X) has the value 128.

9.4.95. MAXLOC (ARRAY [,DIM] [,MASK] [,KIND])

Description: Returns the location of the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array of type integer or real.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY. This argument is a Fortran 95 feature.
MASK (opt)	Must be a logical array that is conformable with ARRAY.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is an array of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if DIM is omitted:

- The array result has rank one and a size equal to the rank of ARRAY.
- If MAXLOC (ARRAY) is specified, the elements in the array result form the subscript of the location of the element with the maximum value in ARRAY. The *i*th subscript returned is in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY.
- If MAXLOC (ARRAY, MASK=MASK) is specified, the elements in the array result form the subscript of the location of the element with the maximum value corresponding to the condition specified by MASK.

The following rules apply if DIM is specified:

- The array result has a rank that is one less than ARRAY, and shape (d_1, d_2,...,d_{DIM-1}, d_{DIM+1},..., d_n), where (d_1, d_2,..., d_n) is the shape of ARRAY.

- If ARRAY has rank one, MAXLOC (ARRAY, DIM [,MASK]) has a value equal to that of MAXLOC (ARRAY [,MASK = MASK]). Otherwise, the value of element ($s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n$) of MAXLOC (ARRAY, DIM [,MASK]) is equal to MAXLOC (ARRAY ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$) [,MASK = MASK ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$)]).

If more than one element has maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, or every element of MASK has the value .FALSE., the value of the result is undefined.

Examples

The value of MAXLOC ((/3, 7, 4, 7/)) is (2), which is the subscript of the location of the first occurrence of the maximum value in the rank-one array.

A is the array $\begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$.

MAXLOC (A, MASK=A .LT. 5) has the value (1, 1) because these are the subscripts of the location of the maximum value (4) that is less than 5.

MAXLOC (A, DIM=1) has the value (1, 2, 3, 2). 1 is the subscript of the location of the maximum value (4) in column 1; 2 is the subscript of the location of the maximum value (1) in column 2; and so forth.

MAXLOC (A, DIM=2) has the value (1, 4, 3). 1 is the subscript of the location of the maximum value in row 1; 4 is the subscript of the location of the maximum value in row 2; and so forth.

9.4.96. MAXVAL (ARRAY [,DIM] [,MASK])

Description:	Returns the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.	
Class:	Transformational function; Generic	
Arguments:	ARRAY	Must be an array of type integer or real.
	DIM (opt)	Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of ARRAY.
	MASK (opt)	Must be a logical array that is conformable with ARRAY.
Results:	The result is an array or a scalar of the same data type as ARRAY.	

The result is a scalar if DIM is omitted or ARRAY has rank one.

The following rules apply if DIM is omitted:

- If MAXVAL (ARRAY) is specified, the result has a value equal to the maximum value of all the elements in ARRAY.
- If MAXVAL (ARRAY, MASK=MASK) is specified, the result has a value equal to the maximum value of the elements in ARRAY corresponding to the condition specified by MASK.

The following rules apply if DIM is specified:

- An array result has a rank that is one less than ARRAY, and shape ($d_1, d_2,...,d_{DIM-1}, d_{DIM+1},..., d_n$), where ($d_1, d_2,..., d_n$) is the shape of ARRAY.
- If ARRAY has rank one, MAXVAL (ARRAY, DIM [,MASK]) has a value equal to that of MAXVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element ($s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n$) of MAXVAL (ARRAY, DIM, [,MASK]) is equal to MAXVAL (ARRAY ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$) [,MASK = MASK ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$)]).

If ARRAY has size zero or if there are no true elements in MASK, the result (if DIM is omitted), or each element in the result array (if DIM is specified), has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind parameters of ARRAY.

Examples

The value of MAXVAL ((/2, 3, 4/)) is 4 because that is the maximum value in the rank-one array.

MAXVAL (B, MASK=B .LT. 0.0) finds the maximum value of the negative elements of B.

C is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$.

MAXVAL (C, DIM=1) has the value (5, 6, 7). 5 is the maximum value in column 1; 6 is the maximum value in column 2; and so forth.

MAXVAL (C, DIM=2) has the value (4, 7). 4 is the maximum value in row 1 and 7 is the maximum value in row 2.

9.4.97. MERGE (TSOURCE, FSOURCE, MASK)

Description:	Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.	
Class:	Elemental function; Generic	
Arguments:	TSOURCE	Must be a scalar or array (of any data type).
	FSOURCE	Must be a scalar or array of the same type and type parameters as TSOURCE.
	MASK	Must be a logical array.
Results:	The result type is the same as TSOURCE. The value of MASK determines whether the result value is taken from TSOURCE (if MASK is true) or FSOURCE (if MASK is false).	

Examples

For MERGE (1.0, 0.0, R < 0), if R is –3, the merge has the value 1.0, while if R is 7, the merge has the value 0.0.

TSOURCE is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, FSOURCE is the array $\begin{bmatrix} 8 & 9 & 0 \\ 1 & 2 & 3 \end{bmatrix}$, and MASK is the array $\begin{bmatrix} F & T & T \\ T & T & F \end{bmatrix}$.

MERGE (TSOURCE, FSOURCE, MASK) produces the result: $\begin{bmatrix} 8 & 3 & 5 \\ 2 & 4 & 3 \end{bmatrix}$.

9.4.98. MIN (A1, A2 [,A3,...])

Description: Returns the minimum value of the arguments.

Class: Elemental function; Generic

Arguments: A1, A2, and A3 (opt) must all have the same type (integer or real) and kind parameters.

Results: For MIN0, AMIN1, DMIN1, QMIN1, IMIN0, JMIN0, and KMIN0, the result type is the same as the arguments. For MIN1, IMIN1, JMIN1, and KMIN1, the result is of type integer. For AMIN0, AIMIN0, AJMIN0, and AKMIN0, the result is of type real. The value of the result is that of the smallest argument.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
	INTEGER(1)	REAL(4)
IMIN0	INTEGER(2)	INTEGER(2)
AIMIN0	INTEGER(2)	REAL(4)
MIN0[2]	INTEGER(4)	INTEGER(4)
AMIN0[4]	INTEGER(4)	REAL(4)
KMIN0	INTEGER(8)	INTEGER(8)
AKMIN0	INTEGER(8)	REAL(4)
IMIN1	REAL(4)	INTEGER(2)
MIN1[6]	REAL(4)	INTEGER(4)
KMIN1	REAL(4)	INTEGER(8)
AMIN1[7]	REAL(4)	REAL(4)
DMIN1	REAL(8)	REAL(8)
QMIN1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMIN0.

[4]In Fortran 95/90, AMIN0 and MIN1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[6]The setting of compiler options specifying integer size can affect MIN1.

[7]The setting of compiler options specifying real size can affect AMIN1.

Examples

MIN (2.0, –8.0, 6.0) has the value –8.0.

MIN (14, 32, –50) has the value –50.

9.4.99. MINEXPONENT (X)

Description: Returns the minimum exponent in the model representing the same type and kind parameters as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type real; it can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value e_{min}, as defined in Section D.2.

Examples

If X is of type REAL(4), MINEXPONENT (X) has the value –125.

9.4.100. MINLOC (ARRAY [,DIM] [,MASK] [,KIND])

Description: Returns the location of the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array of type integer or real.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY. This argument is a Fortran 95 feature.
MASK (opt)	Must be a logical array that is conformable with ARRAY.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is an array of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if DIM is omitted:

- The array result has rank one and a size equal to the rank of ARRAY.
- If MINLOC (ARRAY) is specified, the elements in the array result form the subscript of the location of the element with the minimum value in ARRAY. The *i*th subscript returned is in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY.
- If MINLOC (ARRAY, MASK=MASK) is specified, the elements in the array result form the subscript of the location of the element with the minimum value corresponding to the condition specified by MASK.

The following rules apply if DIM is specified:

- The array result has a rank that is one less than ARRAY, and shape $(d_1, d_2,...,d_{DIM-1}, d_{DIM+1},..., d_n)$, where $(d_1, d_2,..., d_n)$ is the shape of ARRAY.
- If ARRAY has rank one, MINLOC (ARRAY, DIM [,MASK]) has a value equal to that of MINLOC (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n)$ of MINLOC (ARRAY, DIM [,MASK]) is equal to MINLOC (ARRAY $(s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n)$ [,MASK = MASK $(s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n)$]).

If more than one element has minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, or every element of MASK has the value .FALSE., the value of the result is undefined.

Examples

The value of MINLOC ((/3, 1, 4, 1/)) is (2), which is the subscript of the location of the first occurrence of the minimum value in the rank-one array.

A is the array $\begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$.

MINLOC (A, MASK=A .GT. –5) has the value (3, 2) because these are the subscripts of the location of the minimum value (–4) that is greater than –5.

MINLOC (A, DIM=1) has the value (3, 3, 1, 3). 3 is the subscript of the location of the minimum value (–1) in column 1; 3 is the subscript of the location of the minimum value (–4) in column 2; and so forth.

MINLOC (A, DIM=2) has the value (3, 3, 4). 3 is the subscript of the location of the minimum value (–3) in row 1; 3 is the subscript of the location of the minimum value (–2) in row 2; and so forth.

9.4.101. MINVAL (ARRAY [,DIM] [,MASK])

Description: Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array of type integer or real.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
MASK (opt)	Must be a logical array that is conformable with ARRAY.

Results: The result is an array or a scalar of the same data type as ARRAY.

The result is a scalar if DIM is omitted or ARRAY has rank one.

The following rules apply if DIM is omitted:

- If MINVAL (ARRAY) is specified, the result has a value equal to the minimum value of all the elements in ARRAY.
- If MINVAL (ARRAY, MASK=MASK) is specified, the result has a value equal to the minimum value of the elements in ARRAY corresponding to the condition specified by MASK.

The following rules apply if DIM is specified:

- An array result has a rank that is one less than ARRAY, and shape ($d_1, d_2,...,d_{DIM-1}, d_{DIM+1},..., d_n$), where ($d_1, d_2,..., d_n$) is the shape of ARRAY.
- If ARRAY has rank one, MINVAL (ARRAY, DIM [,MASK]) has a value equal to that of MINVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element ($s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n$) of MINVAL (ARRAY, DIM, [,MASK]) is equal to MINVAL (ARRAY ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$) [,MASK = MASK ($s_1, s_2,..., s_{DIM-1}, :, s_{DIM+1},..., s_n$)]).

If ARRAY has size zero or if there are no true elements in MASK, the result (if DIM is omitted), or each element in the result array (if DIM is specified), has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind parameters of ARRAY.

Examples

The value of MINVAL ((/2, 3, 4/)) is 2 because that is the minimum value in the rank-one array.

The value of MINVAL (B, MASK=B .GT. 0.0) finds the minimum value of the positive elements of B.

C is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$.

MINVAL (C, DIM=1) has the value (2, 3, 4). 2 is the minimum value in column 1; 3 is the minimum value in column 2; and so forth.

MINVAL (C, DIM=2) has the value (2, 5). 2 is the minimum value in row 1 and 5 is the minimum value in row 2.

9.4.102. MOD (A, P)

Description:	Returns the remainder when the first argument is divided by the second argument.	
Class:	Elemental function; Generic	
Arguments:	A	Must be of type integer or real.
	P	Must have the same type and kind parameters as A.
Results:	The result type is the same as A. If P is not equal to zero, the value of the result is A – INT(A ÷P) * P. If P is equal to zero, the result is undefined.	

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IMOD	INTEGER(2)	INTEGER(2)
MOD[1]	INTEGER(4)	INTEGER(4)
KMOD	INTEGER(8)	INTEGER(8)
AMOD[2]	REAL(4)	REAL(4)
DMOD	REAL(8)	REAL(8)
QMOD	REAL(16)	REAL(16)

[1]Or JMOD.

[2]The setting of compiler options specifying real size can affect AMOD.

Examples

MOD (7, 3) has the value 1.

MOD (9, –6) has the value 3.

MOD (–9, 6) has the value –3.

9.4.103. MODULO (A, P)

Description:	Returns the modulo of the arguments.	
Class:	Elemental function; Generic	
Arguments:	A	Must be of type integer or real.
	P	Must have the same type and kind parameters as A.
Results:	The result type is the same as A. The result value depends on the type of A, as follows:	

- If A is of type integer and P is not equal to zero, the value of the result is A – FLOOR(REAL(A) ÷REAL(P)) * P.
- If A is of type real and P is not equal to zero, the value of the result is A – FLOOR(A ÷P) * P.

If P is equal to zero (regardless of the type of A), the result is undefined.

Examples

MODULO (7, 3) has the value 1.

MODULO (9, –6) has the value –3.

MODULO (–9, 6) has the value 3.

9.4.104. MULT_HIGH (I, J)

Description:	Multiplies two 64-bit unsigned integers. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.	
Class:	Elemental function	
Arguments:	I	Must be of type INTEGER(8).
	J	Must be of type INTEGER(8).
Results:	The result is of type INTEGER(8). The result value is the upper (leftmost) 64 bits of the 128-bit unsigned result.	

Examples

Consider the following:

```
        INTEGER(8) I,J,K
        I=2_8**53
        J=2_8**51
        K = MULT_HIGH (I,J)
        PRINT *,I,J,K
        WRITE (6,1000)I,J,K
1000    FORMAT (' ', 3(Z,1X))
        END
```

This example prints the following:

```
      9007199254740992      2251799813685248         1099511627776
         20000000000000           8000000000000             10000000000
```

9.4.105. MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

Description:	Copies a sequence of bits (a bit field) from one location to another.
Class:	Elemental subroutine
Arguments:	There are five arguments[1]:
FROM	Can be of any integer type. It represents the location from which a bit field is transferred.
FROMPOS	Can be of any integer type; it must not be negative. It identifies the first bit position in the field transferred from FROM. FROMPOS + LEN must be less than or equal to BIT_SIZE (FROM).[2]

LEN	Can be of any integer type; it must not be negative. It identifies the length of the field transferred from FROM.
TO	Can be of any integer type, but must have the same kind parameter as FROM. It represents the location to which a bit field is transferred. 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 LEN bits of FROM (starting at FROMPOS) had on entry. [2]
TOPOS	Can be of any integer type; it must not be negative. It identifies the starting position (within TO) for the bits being transferred. TOPOS + LEN must be less than or equal to BIT_SIZE (TO).

[1]FROM, FROMPOS, LEN, and TOPOS are INTENT(IN) arguments; TO is an INTENT(INOUT) argument. For more information on INTENT, see Section 5.10.

[2]The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3. For more information on bit functions, see Section 9.3.3.

You can also use the following specific subroutines:

IMVBITS	All arguments must be INTEGER(2).
JMVBITS	Arguments can be INTEGER(2) or INTEGER(4); at least one must be INTEGER(4).
KMVBITS	Arguments can be INTEGER(2), INTEGER(4), or INTEGER(8); at least one must be INTEGER(8).

Examples

If TO has the initial value of 6, its value after a call to MVBITS with arguments (7, 2, 2, TO, 0) is 5.

9.4.106. MY_PROCESSOR ( )

Description: Returns the identifying number of the calling process. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Inquiry function; Specific

Results: The result is a scalar of type default integer. The result value is the identifying number of the physical processor from which the call is made.

The value is in the range 0 to *n*-1, where *n* is the value returned by NUMBER_OF_PROCESSORS.

This function can only be called from within an EXTRINSIC (HPF_LOCAL) procedure.

9.4.107. NEAREST (X, S)

Description: Returns the nearest different number (representable on the processor) in a given direction.

Class: Elemental function; Generic

Arguments:

X	Must be of type real.
S	Must be of type real and nonzero.

Results: The result type is the same as X. A positive S returns the nearest number in the direction of positive infinity. A negative S goes in the direction of negative infinity.

Examples

If 3.0 and 2.0 are REAL(4) values, NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$, which approximately equals 3.0000002, while NEAREST (3.0, -2.0) has the value $3 - 2^{-22}$, which approximately equals 2.9999998. (For more information on the model for REAL(4), see Section D.2).

9.4.108. NINT (A [,KIND])

Description: Returns the nearest integer to the argument.

Class: Elemental function; Generic

Arguments:

A	Must be of type real.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If A is greater than zero, NINT (A) has the value INT (A + 0.5); if A is less than or equal to zero, NINT (A) has the value INT (A – 0.5).

Specific Name	Argument Type	Result Type
ININT	REAL(4)	INTEGER(2)
NINT[2]	REAL(4)	INTEGER(4)
KNINT	REAL(4)	INTEGER(8)
IIDNNT	REAL(8)	INTEGER(2)
IDNINT[3]	REAL(8)	INTEGER(4)
KIDNNT	REAL(8)	INTEGER(8)
IIQNNT	REAL(16)	INTEGER(2)
IQNINT[4]	REAL(16)	INTEGER(4)
KIQNNT[5]	REAL(16)	INTEGER(8)

[2]The setting of compiler options specifying integer size can affect NINT, IDNINT, and IQNINT.

[3]Or JIDNNT. For compatibility with older versions of Fortran, IDNINT can also be specified as a generic function.

[4]Or JIQNNT. For compatibility with older versions of Fortran, IQNINT can also be specified as a generic function.

[5]This specific function cannot be passed as an actual argument.

Examples

NINT (3.879) has the value 4.

NINT (–2.789) has the value –3.

9.4.109. NOT (I)

Description: Returns the logical complement of the argument.

Class: Elemental function; Generic

Arguments: I must be of type integer.

Results: The result type is the same as I. The result value is obtained by complementing I bit-by-bit according to the following truth table:

```
I
NOT (I)

1 0

0 1
```

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
INOT	INTEGER(2)	INTEGER(2)
JNOT	INTEGER(4)	INTEGER(4)
KNOT	INTEGER(8)	INTEGER(8)

Examples

If I has a value equal to 10101010 (base 2), NOT (I) has the value 01010101 (base 2).

9.4.110. NULL ([MOLD])

Description: Initializes a pointer as disassociated when it is declared. This is a new intrinsic procedure in Fortran 95.

Class: Transformational function; Generic

Arguments: MOLD is optional. If used, it must be a pointer; it can be of any type. Its pointer association status can be associated, disassociated, or undefined. If its status is associated, the target does not have to be defined with a value.

Results: The result type is the same as MOLD (if present); otherwise, it is determined as follows:

```
If NULL () Appears...
Type is Determined From...

On the right side of

 pointer assignment The pointer on the left side

As initialization for an

 object in a declaration The object

As default initialization

 for a component The component

In a structure constructor The corresponding component

As an actual argument The corresponding dummy argument
```

```
In a DATA statement The corresponding pointer object
```

The result is a pointer with disassociated association status.

Examples

Consider the following:

```
INTEGER, POINTER :: POINT1 => NULL()
```

This statement defines the initial association status of POINT1 to be disassociated.

9.4.111. NUMBER_OF_PROCESSORS ([DIM])

Description: Returns the total number of processors (peers) available to the program. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Inquiry function; Specific

Results: The result is a scalar of type default integer. The result value is the total number of processors (peers) available to the program.

For a single-processor workstation, the result value is 1.

9.4.112. NWORKERS ( )

Description: Returns the number of processes executing a routine.

This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is provided for compatibility from Fortran compilers on old OpenVMS systems.

Class: Inquiry function; Specific

Arguments: None.

Results: The result is always 1.

9.4.113. PACK (ARRAY, MASK [,VECTOR])

Description: Takes elements from an array and packs them into a rank-one array under the control of a mask.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array (of any data type).
MASK	Must be of type logical and conformable with ARRAY. It determines which elements are taken from ARRAY.
VECTOR (opt)	Must be a rank-one array with the same type and type parameters as ARRAY. Its size must be at least t, where *t* is the number of true elements in MASK. If MASK is a scalar with value true, VECTOR must have at least as many elements as there are in ARRAY. Elements in VECTOR are used to fill out the result array if there are not enough elements selected by MASK.

Results: The result is a rank-one array with the same type and type parameters as ARRAY. If VECTOR is present, the size of the result is that of VECTOR. Otherwise, the size of the result is the number of true elements in MASK, or the number of elements in ARRAY (if MASK is a scalar with value true).

Elements in ARRAY are processed in array element order to form the result array. Element i of the result is the element of ARRAY that corresponds to the ith true element of MASK. If VECTOR is present and has more elements than there are true values in MASK, any result elements that are empty (because they were not true according to MASK) are set to the corresponding values in VECTOR.

Examples

N is the array $\begin{bmatrix} 0 & 8 & 0 \\ 0 & 0 & 0 \\ 7 & 0 & 0 \end{bmatrix}$.

PACK (N, MASK=N .NE. 0, VECTOR=(/1, 3, 5, 9, 11, 13/)) produces the result (7, 8, 5, 9, 11, 13).

PACK (N, MASK=N .NE. 0) produces the result (7, 8).

9.4.114. POPCNT (I)

Description: Returns the number of 1 bits in an integer.

Class: Elemental function; Generic

Arguments: I must be of type integer.

Results: The result type is the same as I. The result value is the number of 1 bits in the binary representation of the integer I.

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

Examples

If the value of I is B ’0...00011010110 ’, the value of POPCNT(I) is 5.

9.4.115. POPPAR (I)

Description: Returns the parity of an integer.

Class: Elemental function; Generic

Arguments: I must be of type integer.

Results: The result type is the same as I. If there are an odd number of 1 bits in the binary representation of the integer I, the result value is 1. If there are an even number, the result value is zero.

The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

Examples

If the value of I is B ’0...00011010110 ’, the value of POPPAR(I) is 1.

9.4.116. PRECISION (X)

Description: Returns the decimal precision in the model representing real numbers with the same kind parameter as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type real or complex. It can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value INT((DIGITS(X) – 1) * LOG10(RADIX(X))). If RADIX(X) is an integral power of 10, 1 is added to the result.

Examples

If X is a REAL(4) value, PRECISION (X) has the value 6. The value 6 is derived from INT ((24-1) * LOG10 (2.)) = INT (6.92...). For more information on the model for REAL(4), see Section D.2.

9.4.117. PRESENT (A)

Description: Returns whether or not an optional dummy argument is present (has an associated actual argument).

Class: Inquiry function; Generic

Arguments: A must be an optional argument of the current procedure.

Results: The result is a scalar of type default logical. The result is .TRUE. if A is present; otherwise, the result is .FALSE..

Examples

Consider the following:

```
SUBROUTINE CHECK (X, Y)
  REAL X, Z
  REAL, OPTIONAL :: Y
  ...
  IF (PRESENT (Y)) THEN
    Z = Y
  ELSE
     Z = X * 2
  END IF
END
...
CALL CHECK (15.0, 12.0)      ! Causes B to be set to 12.0
CALL CHECK (15.0)            ! Causes B to be set to 30.0
```

For more information, including a full example, see Section 8.8.1.1.

9.4.118. PROCESSORS_SHAPE ( )

Description: Returns the shape of an implementation-dependent hardware processor array.

Alpha MPI clusters are one-dimensional processor arrays whose shape is the number of peers.

PROCESSORS_SHAPE is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Inquiry function; Specific

Arguments: None.

Results: If a program is compiled for an Alpha MPI cluster, the result is an array of rank one containing the number of processors (peers) available to the program. Otherwise, the result is always a rank-one array of size zero.

9.4.119. PRODUCT (ARRAY [,DIM] [,MASK])

Description: Returns the product of all the elements in an entire array or in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array of type integer or real.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
MASK (opt)	Must be of type logical and conformable with ARRAY.

Results: The result is an array or a scalar of the same data type as ARRAY.

The result is a scalar if DIM is omitted or ARRAY has rank one.

The following rules apply if DIM is omitted:

- If PRODUCT (ARRAY) is specified, the result is the product of all elements of ARRAY. If ARRAY has size zero, the result is 1.
- If PRODUCT (ARRAY, MASK=MASK) is specified, the result is the product of all elements of ARRAY corresponding to true elements of MASK. If ARRAY has size zero, or every element of MASK has the value .FALSE., the result is 1.

The following rules apply if DIM is specified:

- If ARRAY has rank one, the value is the same as PRODUCT (ARRAY [,MASK=MASK]).
- An array result has a rank that is one less than ARRAY, and shape ($d_1, d_2,...,d_{DIM-1}, d_{DIM+1},..., d_n$), where ($d_1, d_2,..., d_n$) is the shape of ARRAY.
- The value of element ($s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n$) of PRODUCT (ARRAY, DIM [,MASK]) is equal to PRODUCT (ARRAY ($s_1, s_2,..., s_{DIM-1}$, :, $s_{DIM+1},..., s_n$) [,MASK=MASK ($s_1, s_2,..., s_{DIM-1}$, :, $s_{DIM+1},..., s_n$)]).

Examples

PRODUCT ((/2, 3, 4/)) returns the value 24 (the product of 2 * 3 * 4). PRODUCT ((/2, 3, 4/), DIM=1) returns the same result.

PRODUCT (C, MASK=C .LT. 0.0) returns the product of the negative elements of C.

A is the array $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 3 & 5 \end{bmatrix}$.

PRODUCT (A, DIM=1) returns the value (2, 12, 35), which is the product of all elements in each column. 2 is the product of 1 * 2 in column 1. 12 is the product of 4 * 3 in column 2, and so forth.

PRODUCT (A, DIM=2) returns the value (28, 30), which is the product of all elements in each row. 28 is the product of 1 * 4 * 7 in row 1. 30 is the product of 2 * 3 * 5 in row 2.

9.4.120. QCMPLX (X [,Y])

Description: Converts the argument to COMPLEX(16) type. This function must not be passed as an actual argument.

Class: Elemental function; Generic

Arguments:

X	Must be of type integer, real, or complex.
Y (opt)	Must be of type integer or real. It must not be present if X is of type complex.

Results: The result is of type COMPLEX(16) (or COMPLEX*32).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If Y is not specified and X is complex, the result value is CMPLX (REAL(X), AIMAG(X)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

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).

QCMPLX (4.1, 2.3) has the value (4.1Q0, 2.3Q0).

9.4.121. QEXT (A)

Description: Converts a number to quad precision (REAL(16)) type.

Class: Elemental function; Generic

Arguments: A must be of type integer, real, or complex.

Results: The result is of type REAL(16) (REAL*16). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If A is of type REAL(16), the result is the value of the A with no conversion (QEXT(A) = A).

If A is of type integer or real, the result has as much precision of the significant part of A as a REAL(16) value 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 REAL(16) value can contain.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(16)
	INTEGER(2)	REAL(16)

Specific Name[1]	Argument Type	Result Type
	INTEGER(4)	REAL(16)
	INTEGER(8)	REAL(16)
QEXT	REAL(4)	REAL(16)
QEXTD	REAL(8)	REAL(16)
	REAL(16)	REAL(16)
	COMPLEX(4)	REAL(16)
	COMPLEX(8)	REAL(16)
	COMPLEX(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

Examples

QEXT (4) has the value 4.0 (rounded; there are 32 places to the right of the decimal point).

QEXT ((3.4, 2.0)) has the value 3.4 (rounded; there are 32 places to the right of the decimal point).

9.4.122. QFLOAT (A)

Description: Converts an integer to quad precision (REAL(16)) type.

Class: Elemental function; Generic

Arguments: A must be of type integer.

Results: The result is of type REAL(16) (REAL*16).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Examples

QFLOAT (–4) has the value –4.0 (rounded; there are 32 places to the right of the decimal point).

9.4.123. QREAL (A)

Description: Converts the real part of a COMPLEX(16) argument to REAL(16) type. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Elemental function; Specific

Arguments: A must be of type COMPLEX(16) (or COMPLEX*32).

Results: The result is of type REAL(16) (or REAL*16).

Examples

QREAL ((2.0q0, 3.0q0)) has the value 2.0q0.

9.4.124. RADIX (X)

Description: Returns the base of the model representing numbers of the same type and kind parameters as the argument.

Class:	Inquiry function; Generic
Arguments:	X must be of type integer or real; it can be scalar or array valued.
Results:	The result is a scalar of type default integer. For an integer argument, the result has the value r (as defined in Section D.1). For a real argument, the result has the value b (as defined in Section D.2).

Examples

If X is a REAL(4) value, RADIX (X) has the value 2.

9.4.125. RAN (I)

Description:	Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.
Class:	Nonelemental function; Specific
Arguments:	I is the *seed*. It must be an INTEGER(4) variable or array element. It should initially be set to a large, odd integer value. The RAN function stores a value in the argument that is later used to calculate the next random number. There are no restrictions on the seed, although it should be initialized with different values on separate runs to obtain different random numbers.
Results:	The result is of type REAL(4). The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. It is set equal to the value associated with the argument I.

Examples

In RAN (I), if variable I has the value 3, RAN has the value 4.8220158E–05.

9.4.126. RANDOM_NUMBER (HARVEST)

Description:	Returns one pseudorandom number or an array of such numbers.
Class:	Subroutine
Arguments:	HARVEST must be of type real. It is an INTENT(OUT) argument (see Section 5.10), and can be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution within the range $0 <= x < 1$.

Examples

Consider the following:

```
REAL Y, Z (5, 5)
! Initialize Y with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = Y)
CALL RANDOM_NUMBER (Z)
```

Y and Z contain uniformly distributed random numbers.

9.4.127. RANDOM_SEED ([SIZE] [,PUT] [,GET])

Description: Changes or queries the seed (starting point) for the pseudorandom number generator used by RANDOM_NUMBER.

Class: Subroutine

Arguments: No more than one argument can be specified. If no argument is specified, a random number based on the date and time is assigned to the seed. The three optional arguments follow[1]:

SIZE (opt) Must be scalar and of type default integer. It is set to the number of integers (N) that the processor uses to hold the value of the seed.

PUT (opt) Must be a default integer array of rank one and size >= N. It is used to reset the value of the seed.

GET (opt) Must be a default integer array of rank one and size >= N. It is set to the current value of the seed.

[1]SIZE and GET are INTENT(OUT) arguments; PUT is an INTENT(IN) argument. For more information on INTENT, see Section 5.10.

Examples

Consider the following:

```
CALL RANDOM_SEED ( )                    ! Processor reinitializes the
                                        !   seed randomly from the date
                                        !   and time
CALL RANDOM_SEED (SIZE = M)             ! Sets M to N
CALL RANDOM_SEED (PUT = SEED (1 : M))   ! Sets user seed
CALL RANDOM_SEED (GET = OLD  (1 : M))   ! Reads current seed
```

9.4.128. RANDU (I1, I2, X)

Description: Computes a pseudorandom number as a single-precision value.

Class: Subroutine

Arguments:

I1, I2 — INTEGER(2) variables or array elements that contain the *seed* for computing the random number. These values are updated during the computation so that they contain the updated seed.

X — A REAL(4) variable or array element where the computed random number is returned.

Results: The result is returned in X, which must be of type REAL(4). The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for I1 and I2.

If I1=0 and I2=0, the generator base is set as follows:

```
X(n + 1) = 2**16 + 3
```

Otherwise, it is set as follows:

```
X(n + 1) = (2**16 + 3) * X(n) mod 2**32
```

The generator base `X(n + 1)` is stored in I1, I2. The result is `X(n + 1)` scaled to a real value `Y(n + 1)`, for 0.0 <= `Y(n + 1) < 1`.

Examples

Consider the following:

```
REAL X
INTEGER(2) I, J
...
CALL RANDU (I, J, X)
```

If I and J are values 4 and 6, X stores the value 5.4932479E–04.

9.4.129. RANGE (X)

Description: Returns the decimal exponent range in the model representing numbers with the same kind parameter as the argument.

Class: Inquiry function; Generic

Arguments: X must be of type integer, real, or complex. It can be scalar or array valued.

Results: The result is a scalar of type default integer.

For an integer argument, the result has the value INT (LOG10 (HUGE(X))). For information on the integer model, see Section D.1; on HUGE, see Section 9.4.55.

For a real or complex argument, the result has the value INT(MIN (LOG10(HUGE(X)), –LOG10(TINY(X)))). For information on the real model, see Section D.2; on TINY, see Section 9.4.156.

Examples

If X is a REAL(4) value, RANGE (X) has the value 37. (HUGE(X) = $(1 - 2^{-24}) \times 2^{128}$ and TINY(X) = 2^{-126}).

9.4.130. REAL (A [,KIND])

Description: Converts a value to real type.

Class: Elemental function; Generic

Arguments:

A	Must be of type integer, real, or complex.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type real. If KIND is present, the kind parameter is that specified by KIND. If KIND is not present, see the following table for the kind parameter.

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

If A is integer or real, the result is equal to an approximation of A. If A is complex, the result is equal to an approximation of the real part of A.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(4)
FLOATI	INTEGER(2)	REAL(4)
FLOAT[23]	INTEGER(4)	REAL(4)
REAL[3]	INTEGER(4)	REAL(4)

Specific Name[1]	Argument Type	Result Type
FLOATK	INTEGER(8) For compatibility with older versions of Fortran, FLOAT can also be specified as a generic function	REAL(4)
	REAL(4)	REAL(4)
SNGL[23]	REAL(8)	REAL(4)
SNGLQ	REAL(16)	REAL(4)
	COMPLEX(4)	REAL(4)
	COMPLEX(8)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

[2]Or FLOATJ. For compatibility with older versions of Fortran, FLOAT can also be specified as a generic function.

[3]The setting of compiler options specifying real size can affect FLOAT, REAL, and SNGL.

Examples

REAL (–4) has the value –4.0.

REAL (Y) has the same kind parameter and value as the real part of complex variable Y.

9.4.131. REPEAT (STRING, NCOPIES)

Description:	Concatenates several copies of a string.	
Class:	Transformational function; Generic	
Arguments:	STRING	Must be scalar and of type character.
	NCOPIES	Must be scalar and of type integer. It must not be negative.
Results:	The result is a scalar of type character and length NCOPIES × LEN(STRING). The kind parameter is the same as STRING. The value of the result is the concatenation of NCOPIES copies of STRING.	

Examples

REPEAT ('S', 3) has the value SSS.

REPEAT ('ABC', 0) has the value of a zero-length string.

9.4.132. RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])

Description:	Constructs an array with a different shape from the argument array.	
Class:	Transformational function; Generic	
Arguments:	SOURCE	Must be an array (of any data type). It supplies the elements for the result array. Its size must be greater than or equal to PRODUCT(SHAPE) if PAD is omitted or has size zero.
	SHAPE	Must be an integer array of up to 7 elements, with rank one and constant size. It defines the shape of the result

		array. Its size must be positive; its elements must not have negative values.
	PAD (opt)	Must be an array with the same type and kind parameters as SOURCE. It is used to fill in extra values if the result array is larger than SOURCE.
	ORDER (opt)	Must be an integer array with the same shape as SHAPE. Its elements must be a permutation of (1,2,...,n), where *n* is the size of SHAPE. If ORDER is omitted, it is assumed to be (1,2,...,n).

Results: The result is an array of shape SHAPE with the same type and kind parameters as SOURCE. The size of the result is the product of the values of the elements of SHAPE.

In the result array, the array elements of SOURCE are placed in the order of dimensions specified by ORDER. If ORDER is omitted, the array elements are placed in normal array element order.

The array elements of SOURCE are followed (if necessary) by the array elements of PAD in array element order. If necessary, additional copies of PAD follow until all the elements of the result array have values.

Examples

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 3/)) has the value $\begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{bmatrix}$.

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value $\begin{bmatrix} 3 & 4 & 5 & 6 \\ 7 & 8 & 1 & 1 \end{bmatrix}$.

9.4.133. RRSPACING (X)

Description: Returns the reciprocal of the relative spacing of model numbers near the argument value.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X. The result has the value $|X * b^{-e}| \times b^{p}$. Parameters *b*, *e*, *p* are defined in Section D.2.

Examples

If –3.0 is a REAL(4) value, RRSPACING (–3.0) has the value 0.75×2^{24}.

9.4.134. SCALE (X, I)

Description: Returns the value of the exponent part (of the model for the argument) changed by a specified value.

Class: Elemental function; Generic

Arguments:	X	Must be of type real.
	I	Must be of type integer.

Results: The result type is the same as X. The result has the value $X * b^{I}$. Parameter *b* is defined in Section D.2.

Examples

If 3.0 is a REAL(4) value, SCALE (3.0, 2) has the value 12.0 and SCALE (3.0, 3) has the value 24.0.

9.4.135. SCAN (STRING, SET [,BACK] [,KIND])

Description:	Scans a string for any character in a set of characters.	
Class:	Elemental function; Generic	
Arguments:	STRING	Must be of type character.
	SET	Must be of type character with the same kind parameter as STRING.
	BACK (opt)	Must be of type logical.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If BACK is omitted (or is present with the value false) and STRING has 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.

If BACK is present with the value true and STRING has 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.

If no character of STRING is in SET or the length of STRING or SET is zero, the value of the result is zero.

Examples

SCAN ('ASTRING', 'ST') has the value 2.

SCAN ('ASTRING', 'ST', BACK=.TRUE.) has the value 3.

SCAN ('ASTRING', 'CD') has the value zero.

9.4.136. SECNDS (X)

Description: Provides the system time of day, or elapsed time, as a floating-point value in seconds.

This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.

Class: Elemental function; Specific

Arguments: X must be of type REAL(4).

Results: The result type is the same as X. The result value is the time in seconds since midnight – X. (The function also produces correct results for time intervals that span midnight.)

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

You can get more precise timing information by using the following Run-Time Library (RTL) procedures:

- LIB$INIT_TIMER
- LIB$SHOW_TIMER
- LIB$STAT_TIMER

Examples

The following shows how to use SECNDS to perform elapsed-time computations:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)

C     CODE TO BE TIMED
      ...
      DELTA = SECNDS(T1)      ! DELTA gives the elapsed time
```

9.4.137. SELECTED_INT_KIND (R)

Description: Returns the value of the kind parameter of an integer data type.

Class: Transformational function; Generic

Arguments: R must be scalar and of type integer.

Results: The result is a scalar of type default integer. The result has a value equal to the value of the kind parameter of the integer data type that represents all values *n* in the range of values *n* with $-10^R < n < 10^R$.

If no such kind type parameter is available on the processor, 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. (For information on the integer model, see Section D.1).

Examples

SELECTED_INT_KIND (6) = 4

9.4.138. SELECTED_REAL_KIND ([P] [,R])

Description: Returns the value of the kind parameter of a real data type.

Class: Transformational function; Generic

Arguments:

P (opt)	Must be scalar and of type integer.
R (opt)	Must be scalar and of type integer.

At least one argument must be specified.

Results: The result is a scalar of type default integer. The result has a value equal to a value of the kind parameter of a real data type with decimal precision, as returned by the

function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R.

If no such kind type parameter is available on the processor, the result is as follows:

–1 if the precision is not available
–2 if the exponent range is not available
–3 if neither is available

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. (For information on the real model, see Section D.2).

Examples

SELECTED_REAL_KIND (6, 70) = 8

9.4.139. SET_EXPONENT (X, I)

Description:	Returns the value the first argument would have if its exponent part were set to the second argument.	
Class:	Elemental function; Generic	
Arguments:	X	Must be of type real.
	I	Must be of type integer.
Results:	The result type is the same as X. The result has the value $X \times b^{I-e}$. Parameters *b* and *e* are defined in Section D.2. If X has the value zero, the result is zero.	

Examples

Assume the following pseudocode:

```
struct FLOAT {
    int exponent;
    float fraction;
}

FLOAT set_exponent( FLOAT x, int i )
{
    FLOAT y = x;
    y.exponent = i;
    return y;
}
```

Note that the operation is performed on the representation of the number, not on the model number. The exponent argument is adjusted for the excess-128 notation used in the exponent field of a floating-point number. The fraction field is not modified.

For example, if X is a REAL*4 variable holding 0.75, that value is represented as 0.75 * 2 ** 0. The exponent is zero. SET_EXPONENT (X, 3) returns 0.75 * 2 ** 3, which is 6.0. SET_EXPONENT (X, 4) returns 0.75 * 2 ** 4, which is 12.0.

9.4.140. SHAPE (SOURCE [,KIND])

Description: Returns the shape of an array or scalar argument.

Class:	Inquiry function; Generic	
Arguments:	SOURCE	Is a scalar or array (of any data type). It must not be an assumed-size array, a disassociated pointer, or an allocatable array that is not allocated.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is a rank-one integer array whose size is equal to the rank of SOURCE. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is the shape of SOURCE.

The setting of compiler options that specify integer size can affect the result of this function.

Examples

SHAPE (2) has the value of a rank-one array of size zero.

If B is declared as B(2:4, –3:1), then SHAPE (B) has the value (3, 5).

9.4.141. SIGN (A, B)

Description:	Returns the absolute value of A times the sign of B.	
Class:	Elemental function; Generic	
Arguments:	A	Must be of type integer or real.
	B	Must have the same type and kind parameters as A.

Results: The result type is the same as A. The value of the result is |A | if B >= zero and -- |A | if B < zero.

If B is of type real and zero, the value of the result is |A |. However, if the processor can distinguish between positive and negative real zero and the appropriate compiler option is specified, the following occurs:

- If B is positive real zero, the value of the result is |A |.
- If B is negative real zero, the value of the result is -- |A |.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IISIGN	INTEGER(2)	INTEGER(2)
ISIGN[1]	INTEGER(4)	INTEGER(4)
KISIGN	INTEGER(8)	INTEGER(8)
SIGN	REAL(4)	REAL(4)
DSIGN	REAL(8)	REAL(8)
QSIGN	REAL(16)	REAL(16)

[1]Or JISIGN. For compatibility with older versions of Fortran, ISIGN can also be specified as a generic function.

Examples

SIGN (4.0, –6.0) has the value –4.0.

SIGN (–5.0, 2.0) has the value 5.0.

9.4.142. SIN (X)

Description: Produces the sine of X.

Class: Elemental function; Generic

Arguments: X must be of type real or complex. It must be in radians and is treated as modulo 2* π. (If X is of type complex, its real part is regarded as a value in radians.)

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
SIN	REAL(4)	REAL(4)
DSIN	REAL(8)	REAL(8)
QSIN	REAL(16)	REAL(16)
CSIN[1]	COMPLEX(4)	COMPLEX(4)
CDSIN[2]	COMPLEX(8)	COMPLEX(8)
CQSIN	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CSIN.

[2]This function can also be specified as ZSIN.

Examples

SIN (2.0) has the value 0.9092974.

SIN (0.8) has the value 0.7173561.

9.4.143. SIND (X)

Description: Produces the sine of X.

Class: Elemental function; Generic

Arguments: X must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
SIND	REAL(4)	REAL(4)
DSIND	REAL(8)	REAL(8)
QSIND	REAL(16)	REAL(16)

Examples

SIND (2.0) has the value 3.4899496E–02.

SIND (0.8) has the value 1.3962180E–02.

9.4.144. SINH (X)

Description: Produces a hyperbolic sine.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
SINH	REAL(4)	REAL(4)
DSINH	REAL(8)	REAL(8)
QSINH	REAL(16)	REAL(16)

Examples

SINH (2.0) has the value 3.626860.

SINH (0.8) has the value 0.8881060.

9.4.145. SIZE (ARRAY [,DIM] [,KIND])

Description: Returns the total number of elements in an array, or the extent of an array along a specified dimension.

Class: Inquiry function; Generic

Arguments:

ARRAY	Must be an array (of any data type). It must not be a disassociated pointer or an allocatable array that is not allocated. It can be an assumed-size array if DIM is present with a value less than the rank of ARRAY.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is a scalar of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If DIM is present, the result is the extent of dimension DIM in ARRAY; otherwise, the result is the total number of elements in ARRAY.

The setting of compiler options that specify integer size can affect the result of this function.

Examples

If B is declared as B(2:4, –3:1), then SIZE (B, DIM=2) has the value 5 and SIZE (B) has the value 15.

9.4.146. SIZEOF (X)

Description: Returns the number of bytes of storage used by the argument. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Class: Inquiry function; Specific

Arguments: X is a scalar or array (of any data type). It must *not* be an assumed-size array.

Results: The result is of type INTEGER(8). The result value is the number of bytes of storage used by X.

Examples

SIZEOF (3.44) has the value 4.

SIZEOF ('SIZE') has the value 4.

9.4.147. SPACING (X)

Description: Returns the absolute spacing of model numbers near the argument value.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X. The result has the value b $^{e-p}$. Parameters *b*, *e*, and *p* are defined in Section D.2. If the result value is outside of the real model range, the result is TINY(X). (For information on TINY, see Section 9.4.156).

Examples

If 3.0 is a REAL(4) value, SPACING (3.0) has the value 2 $^{-22}$.

9.4.148. SPREAD (SOURCE, DIM, NCOPIES)

Description: Creates a replicated array with an added dimension by making copies of existing elements along a specified dimension.

Class: Transformational function; Generic

Arguments:

SOURCE	Must be a scalar or array (of any data type). The rank must be less than 7.
DIM	Must be scalar and of type integer. It must have a value in the range 1 to n + 1 (inclusive), where *n* is the rank of SOURCE.
NCOPIES	Must be scalar and of type integer. It becomes the extent of the additional dimension in the result.

Results: The result is an array of the same type as SOURCE and of rank that is one greater than SOURCE.

If SOURCE is an array, each array element in dimension DIM of the result is equal to the corresponding array element in SOURCE.

If SOURCE is a scalar, the result is a rank-one array with NCOPIES elements, each with the value SOURCE.

If NCOPIES <= zero, the result is an array of size zero.

Examples

SPREAD ("B", 1, 4) is the character array (/"B", "B", "B", "B"/).

B is the array (3, 4, 5) and NC has the value 4.

SPREAD (B, DIM=1, NCOPIES=NC) produces the array $\begin{bmatrix} 3 & 4 & 5 \\ 3 & 4 & 5 \\ 3 & 4 & 5 \\ 3 & 4 & 5 \end{bmatrix}$.

SPREAD (B, DIM=2, NCOPIES=NC) produces the array $\begin{bmatrix} 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 \end{bmatrix}$.

9.4.149. SQRT (X)

Description: Derives the square root of the argument.

Class: Elemental function; Generic

Arguments: X must be of type real or complex. If X is type real, its value must be greater than or equal to zero.

Results: The result type is the same as X. The result has a value equal to the square root of X. A result of type complex is the principal value, with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Specific Name	Argument Type	Result Type
SQRT	REAL(4)	REAL(4)
DSQRT	REAL(8)	REAL(8)
QSQRT	REAL(16)	REAL(16)
CSQRT[1]	COMPLEX(4)	COMPLEX(4)
CDSQRT [2]	COMPLEX(8)	COMPLEX(8)
CQSQRT	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CSQRT.

[2]This function can also be specified as ZSQRT.

Examples

SQRT (16.0) has the value 4.0.

SQRT (3.0) has the value 1.732051.

9.4.150. SUM (ARRAY [,DIM] [,MASK])

Description: Returns the sum of all the elements in an entire array or in a specified dimension of an array.

Class: Transformational function; Generic

Arguments:

ARRAY	Must be an array of type integer, real, or complex.
DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
MASK (opt)	Must be of type logical and conformable with ARRAY.

Results: The result is an array or a scalar of the same data type as ARRAY.

The result is a scalar if DIM is omitted or ARRAY has rank one.

The following rules apply if DIM is omitted:

- If SUM (ARRAY) is specified, the result is the sum of all elements of ARRAY. If ARRAY has size zero, the result is zero.
- If SUM (ARRAY, MASK=MASK) is specified, the result is the sum of all elements of ARRAY corresponding to true elements of MASK. If ARRAY has size zero, or every element of MASK has the value .FALSE., the result is zero.

The following rules apply if DIM is specified:

- If ARRAY has rank one, the value is the same as SUM (ARRAY [,MASK=MASK]).
- An array result has a rank that is one less than ARRAY, and shape ($d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n$), where ($d_1, d_2, ..., d_n$) is the shape of ARRAY.
- The value of element ($s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n$) of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY ($s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$)) [,MASK=MASK ($s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$)].

Examples

SUM ((/2, 3, 4/)) returns the value 9 (sum of 2 + 3 + 4). SUM ((/2, 3, 4/), DIM=1) returns the same result.

SUM (B, MASK=B .LT. 0.0) returns the arithmetic sum of the negative elements of B.

C is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

SUM (C, DIM=1) returns the value (5, 7, 9), which is the sum of all elements in each column. 5 is the sum of 1 + 4 in column 1. 7 is the sum of 2 + 5 in column 2, and so forth.

SUM (C, DIM=2) returns the value (6, 15), which is the sum of all elements in each row. 6 is the sum of 1 + 2 + 3 in row 1. 15 is the sum of 4 + 5 + 6 in row 2.

9.4.151. SYSTEM_CLOCK ([COUNT] [,COUNT_RATE] [,COUNT_MAX])

Description:	Returns integer data from a real-time clock. [1]
Class:	Subroutine
Arguments:	There are three optional arguments[2]:
COUNT (opt)	Must be scalar and of type default integer. It is set to a value based on the current value of the processor clock. The value is increased by one for each clock count until the value COUNT_MAX is reached, and is reset to zero at the next count. (COUNT lies in the range 0 to COUNT_MAX.)
COUNT_ RATE (opt)	Must be scalar and of type default integer. It is set to the number of processor clock counts per second modified by the kind of COUNT_RATE. If default integer is INTEGER(2), COUNT_RATE is 1000. If default integer is INTEGER(4), COUNT_RATE is 10000. If default integer is INTEGER(8), COUNT_RATE is 1000000.

COUNT_MAX (opt)	Must be scalar and of type default integer. It is set to the maximum value that COUNT can have, HUGE(0)[3].

[1]SYSTEM_CLOCK returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. The number is returned with no bias. To get the elapsed time, you must call SYSTEM_CLOCK twice, and subtract the starting time value from the ending time value.

[2]All are INTENT(OUT) arguments. (See Section 5.10.)

[3]For more information on HUGE, see Section 9.4.55.

Examples

Consider the following:

```
INTEGER(2) :: IC2, CRATE2, CMAX2
INTEGER(4) :: IC4, CRATE4, CMAX4
CALL SYSTEM_CLOCK(COUNT=IC2, COUNT_RATE=CRATE2, COUNT_MAX=CMAX2)
CALL SYSTEM_CLOCK(COUNT=IC4, COUNT_RATE=CRATE4, COUNT_MAX=CMAX4)
PRINT *, IC2, CRATE2, CMAX2
PRINT *, IC4, CRATE4, CMAX4
end
```

This program was run on Thursday Dec 11, 1997 at 14:23:55 EST and produced the following output:

```
13880   1000  32767
1129498807       10000  2147483647
```

9.4.152. TAN (X)

Description: Produces the tangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real. It must be in radians and is treated as modulo 2 * π.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
TAN	REAL(4)	REAL(4)
DTAN	REAL(8)	REAL(8)
QTAN	REAL(16)	REAL(16)

Examples

TAN (2.0) has the value –2.185040.

TAN (0.8) has the value 1.029639.

9.4.153. TAND (X)

Description: Produces the tangent of X.

Class: Elemental function; Generic

Arguments: X must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
TAND	REAL(4)	REAL(4)

Specific Name	Argument Type	Result Type
DTAND	REAL(8)	REAL(8)
QTAND	REAL(16)	REAL(16)

Examples

TAND (2.0) has the value 3.4920771E–02.

TAND (0.8) has the value 1.3963542E–02.

9.4.154. TANH (X)

Description: Produces a hyperbolic tangent.

Class: Elemental function; Generic

Arguments: X must be of type real.

Results: The result type is the same as X.

Specific Name	Argument Type	Result Type
TANH	REAL(4)	REAL(4)
DTANH	REAL(8)	REAL(8)
QTANH	REAL(16)	REAL(16)

Examples

TANH (2.0) has the value 0.9640276.

TANH (0.8) has the value 0.6640368.

9.4.155. TIME (BUF)

Description: Returns the current time as set within the system.

Class: Subroutine

Arguments: BUF is an 8-byte variable, array, array element, or character substring.

The date is returned as an 8-byte ASCII character string taking the form hh:mm:ss, where:

hh is the 2-digit hour
mm is the 2-digit minute
ss is the 2-digit second

If BUF is of numeric type and smaller than 8 bytes, data corruption can occur.

If BUF is of character type, its associated length is passed to the subroutine. If BUF is smaller than 8 bytes, the subroutine truncates the date to fit in the specified length. If a CHARACTER array is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Examples

An example of a value returned from a call to TIME is 13:45:23 (a 24-hour clock is used).

Consider the following:

```
CHARACTER*1 HOUR(8)
...
CALL TIME (HOUR)
```

The length of the first array element in CHARACTER array HOUR is passed to the TIME subroutine. The subroutine then truncates the time to fit into the 1-character element, producing an incorrect result.

9.4.156. TINY (X)

Description:	Returns the smallest number in the model representing the same type and kind parameters as the argument.
Class:	Inquiry function; Generic
Arguments:	X must be of type real; it can be scalar or array valued.
Results:	The result is a scalar with the same type and kind parameters as X. The result has the value $b^{e_{min}-1}$. Parameters b and e_{min} are defined in Section D.2.

Examples

If X is of type REAL(4), TINY (X) has the value 2^{-126}.

9.4.157. TRAILZ (I)

Description:	Returns the number of trailing zero bits in an integer.
Class:	Elemental function; Generic
Arguments:	I must be of type integer.
Results:	The result type is the same as I. The result value is the number of trailing zeros in the binary representation of the integer I. The model for the interpretation of an integer value as a sequence of bits is shown in Section D.3.

Examples

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
  TYPE *, TRAILZ(TWO**J)  ! Prints 64, then 0 up to
ENDDO                     !   40 (trailing zeros)
END
```

9.4.158. TRANSFER (SOURCE, MOLD [,SIZE])

Description:	Copies the bit pattern of SOURCE and interprets it according to the type and kind parameters of MOLD.	
Class:	Transformational function; Generic	
Arguments:	SOURCE	Must be a scalar or array (of any data type).

	MOLD	Must be a scalar or array (of any data type). It provides the type characteristics (not a value) for the result.
	SIZE (opt)	Must be scalar and of type integer. It provides the number of elements for the output result.

Results: The result has the same type and type parameters as MOLD.

If MOLD is a scalar and SIZE is omitted, the result is a scalar.

If MOLD is an array and SIZE is omitted, the result is a rank-one array. Its size is the smallest that is possible to hold all of SOURCE.

If SIZE is present, the result is a rank-one array of size SIZE.

If the physical representation of the result is larger than SOURCE, the result contains SOURCE's bit pattern in its right-most bits; the left-most bits of the result are undefined.

If the physical representation of the result is smaller than SOURCE, the result contains the right-most bits of SOURCE's bit pattern.

Examples

TRANSFER (1082130432, 0.0) has the value 4.0 (on processors that represent the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000).

TRANSFER ((/2.2, 3.3, 4.4/), ((0.0, 0.0))) results in a scalar whose value is (2.2, 3.3).

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/)) results in a complex rank-one array of length 2. Its first element is (2.2,3.3) and its second element has a real part with the value 4.4 and an undefined imaginary part.

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/), 1) results in a complex rank-one array having one element with the value (2.2, 3.3).

9.4.159. TRANSPOSE (MATRIX)

Description: Transposes an array of rank two.

Class: Transformational function; Generic

Arguments: MATRIX must be a rank-two array (of any data type).

Results: The result is a rank-two array with the same type and kind parameters as MATRIX. Its shape is (n, m), where (m, n) is the shape of MATRIX. For example, if the shape of MATRIX is (4,6), the shape of the result is (6,4).

Element (i, j) of the result has the value MATRIX (j, i), where *i* is in the range 1 to n, and *j* is in the range 1 to m.

Examples

B is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \end{bmatrix}$.

TRANSPOSE (B) has the value $\begin{bmatrix} 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 1 \end{bmatrix}$.

9.4.160. TRIM (STRING)

Description:	Returns the argument with trailing blanks removed.
Class:	Transformational function; Generic
Arguments:	STRING must be a scalar of type character.
Results:	The result is of type character with the same kind parameter as STRING. Its length is the length of STRING minus the number of trailing blanks in STRING. The value of the result is the same as STRING, except any trailing blanks are removed. If STRING contains only blank characters, the result has zero length.

Examples

TRIM (ΔΔNAMEΔΔΔΔ') has the value 'ΔΔNAME'.

TRIM ('ΔΔCΔΔDΔΔΔΔΔ') has the value 'ΔΔCΔΔD'.

9.4.161. UBOUND (ARRAY [,DIM] [,KIND])

Description:	Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.	
Class:	Inquiry function; Generic	
Arguments:	ARRAY	Must be an array (of any data type). It must not be an allocatable array that is not allocated, or a disassociated pointer. It can be an assumed-size array if DIM is present with a value less than the rank of ARRAY.
	DIM (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of ARRAY.
	KIND (opt)	Must be a scalar integer initialization expression.
Results:	The result type is integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined. If DIM is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of ARRAY. Each element in the result corresponds to a dimension of ARRAY. If ARRAY is an array section or an array expression that is not a whole array or array structure component, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension. If ARRAY is a whole array or array structure component, UBOUND (ARRAY, DIM) has a value equal to the upper bound for subscript DIM of ARRAY (if DIM is nonzero). If DIM has size zero, the corresponding element of the result has the value zero. The setting of compiler options that specify integer size can affect the result of this function.	

Examples

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

UBOUND (ARRAY_A) is (3, 8). UBOUND (ARRAY_A, DIM=2) is 8.

UBOUND (ARRAY_B) is (8, 20). UBOUND (ARRAY_B (5:8, :)) is (4,24) because the number of elements is significant for array section arguments.

9.4.162. UNPACK (VECTOR, MASK, FIELD)

Description:	Takes elements from a rank-one array and unpacks them into another (possibly larger) array under the control of a mask.	
Class:	Transformational function; Generic	
Arguments:	VECTOR	Must be a rank-one array (of any data type). Its size must be at least t, where *t* is the number of true elements in MASK.
	MASK	Must be a logical array. It determines where elements of VECTOR are placed when they are unpacked.
	FIELD	Must be of the same type and type parameters as VECTOR and conformable with MASK. Elements in FIELD are inserted into the result array when the corresponding MASK element has the value false.
Results:	The result is an array with the same shape as MASK, and the same type and type parameters as VECTOR.	
	Elements in the result array are filled in array element order. If element i of MASK is true, the corresponding element of the result is filled by the next element in VECTOR. Otherwise, it is filled by FIELD (if FIELD is scalar) or the ith element of FIELD (if FIELD is an array).	

Examples

N is the array $\begin{bmatrix}0 & 0 & 1\\1 & 0 & 1\\1 & 0 & 0\end{bmatrix}$, P is the array (2, 3, 4, 5), and Q is the array $\begin{bmatrix}T & F & F\\F & T & F\\T & T & F\end{bmatrix}$.

UNPACK (P, MASK=Q, FIELD=N) produces the result $\begin{bmatrix}2 & 0 & 1\\1 & 4 & 1\\3 & 5 & 0\end{bmatrix}$.

UNPACK (P, MASK=Q, FIELD=1) produces the result $\begin{bmatrix}2 & 1 & 1\\1 & 4 & 1\\3 & 5 & 1\end{bmatrix}$.

9.4.163. VERIFY (STRING, SET [,BACK] [,KIND])

Description:	Verifies that a set of characters contains all the characters in a string by identifying the first character in the string that is not in the set.
Class:	Elemental function; Generic

Arguments:	STRING	Must be of type character.
	SET	Must be of type character with the same kind parameter as STRING.
	BACK (opt)	Must be of type logical.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If BACK is omitted (or is present with the value false) and STRING has 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.

If BACK is present with the value true and STRING has 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.

If each character of STRING is in SET or the length of STRING is zero, the value of the result is zero.

Examples

VERIFY ('CDDDC', 'C') has the value 2.

VERIFY ('CDDDC', 'C', BACK=.TRUE.) has the value 4.

VERIFY ('CDDDC', 'CD') has the value zero.

9.4.164. ZEXT (X [,KIND])

Description: Extends the argument with zeros. This function is used primarily for bit-oriented operations.

Class: Elemental function; Generic

Arguments:	X	Must be of type logical or integer.
	KIND (opt)	Must be a scalar integer initialization expression.

Results: The result is of type integer. If KIND is present, the kind parameter of the result is that specified by KIND; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value is X extended with zeros and treated as an unsigned value.

The storage requirements for integer constants are never less than two bytes. Integer constants within the range of constants that can be represented by a single byte still require two bytes of storage.

The setting of compiler options specifying integer size can affect ZEXT.

Specific Name	Argument Type	Result Type
IZEXT	LOGICAL(1)	INTEGER(2)

Specific Name	Argument Type	Result Type
	LOGICAL(2)	INTEGER(2)
	INTEGER(1)	INTEGER(2)
	INTEGER(2)	INTEGER(2)
JZEXT	LOGICAL(1)	INTEGER(4)
	LOGICAL(2)	INTEGER(4)
	LOGICAL(4)	INTEGER(4)
	INTEGER(1)	INTEGER(4)
	INTEGER(2)	INTEGER(4)
	INTEGER(4)	INTEGER(4)
KZEXT	LOGICAL(1)	INTEGER(8)
	LOGICAL(2)	INTEGER(8)
	LOGICAL(4)	INTEGER(8)
	LOGICAL(8)	INTEGER(8)
	INTEGER(1)	INTEGER(8)
	INTEGER(2)	INTEGER(8)
	INTEGER(4)	INTEGER(8)
	INTEGER(8)	INTEGER(8)

Examples

Consider the following example:

```
INTEGER(2) W_VAR /'FFFF'X/
INTEGER(4) L_VAR
L_VAR = ZEXT(W_VAR)
```

This example stores an INTEGER(2) quantity in the low-order 16 bits of an INTEGER(4) quantity, with the resulting value of L_VAR being '0000FFFF'X. If the ZEXT function had not been used, the resulting value would have been 'FFFFFFFF'X, because W_VAR would have been converted to the left-hand operand's data type by sign extension.

Chapter 10. Data Transfer I/O Statements

10.1. Overview of Records and Files

A record is a sequence of values or a sequence of characters. There are three kinds of Fortran records:

- Formatted

  A record containing formatted data that requires translation from internal to external form. Formatted I/O statements have explicit format specifiers (which can specify list-directed formatting) or namelist specifiers (for namelist formatting). Only formatted I/O statements can read formatted data.

- Unformatted

  A record containing unformatted data that is not translated from internal form. An unformatted record can also contain no data. The internal representation of unformatted data is processor-dependent. Only unformatted I/O statements can read unformatted data.

- Endfile

  The last record of a file. An endfile record can be explicitly written to a sequential file by an ENDFILE statement (see Section 12.4 for details).

A file is a sequence of records. There are two types of Fortran files, as follows:

- External

  A file that exists in a medium (such as computer disks or terminals) external to the executable program.

  Records in an external file must be either all formatted or all unformatted. There are three ways to access records in external files: sequential, keyed access, and direct access.

  In sequential access, records are processed in the order in which they appear in the file. In direct access, records are selected by record number, so they can be processed in any order. In keyed access, records are processed by key-field value.

- Internal

  Memory (internal storage) that behaves like a file. This type of file provides a way to transfer and convert data in memory from one format to another. The contents of these files are stored as scalar character variables.

For More Information:

On formatted and unformatted data transfers and external file access methods, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.2. Components of Data Transfer Statements

Data transfer statements take one of the following forms:

```
io-keyword (io-control-list) [io-list]
io-keyword format [,io-list]
```

`io-keyword`

Is one of the following: ACCEPT, PRINT (or TYPE), READ, REWRITE, or WRITE.

`io-control-list`

Is one or more of the following input/output (I/O) control specifiers:

[UNIT=]io-unit	ADVANCE	ERR	KEYID
[FMT=]format	END	IOSTAT	REC
[NML=]group	EOR	KEY[con]	SIZE

`io-list`

Is an I/O list, which can contain variables (except for assumed-size arrays) or implied-do lists. Output statements can contain constants or expressions.

`format`

Is the nonkeyword form of a control-list format specifier (no FMT=).

If a format specifier ([FMT=]format) or namelist specifier ([NML=]group) is present, the data transfer statement is called a formatted I/O statement; otherwise, it is an unformatted I/O statement.

If a record specifier (REC=) is present, the data transfer statement is a direct-access I/O statement; otherwise, it is a sequential-access I/O statement.

If an error, end-of-record, or end-of-file condition occurs during data transfer, file positioning and execution are affected, and certain control-list specifiers (if present) become defined. (For more information, see Section 10.2.1.8.)

Section 10.2.1 describes the I/O control list and Section 10.2.2 describes I/O lists.

10.2.1. I/O Control List

The I/O control list specifies one or more of the following:

- The I/O unit to act upon ([UNIT=]io-unit)

  This specifier must be present; the rest are optional.

- The format (explicit or list-directed) to use for data editing; if explicit, the keyword form must appear ([FMT=]format)

- The namelist group name to act upon ([NML=]group)

- The number of a record to access (REC)
- The name of a variable that contains the completion status of an I/O operation (IOSTAT)
- The label of the statement that receives control if an error (ERR), end-of-file (END), or end-of-record (EOR) condition occurs
- The key field (KEY[con]) and key of reference (KEYID) to access a keyed-access record
- Whether you want to use advancing or nonadvancing I/O (ADVANCE)
- The number of characters read from a record (SIZE) by a nonadvancing READ statement

No control specifier can appear more than once, and the list must not contain both a format specifier and namelist group name specifier.

Control specifiers can take any of the following forms:

- Keyword form

  When the keyword form (for example, UNIT=io-unit) is used for all control-list specifiers in an I/O statement, the specifiers can appear in any order.

- Nonkeyword form

  When the nonkeyword form (for example, io-unit) is used for all control-list specifiers in an I/O statement, the io-unit specifier must be the first item in the control list. If a format specifier or namelist group name specifier is used, it must immediately follow the io-unit specifier.

- Mixed form

  When a mix of keyword and nonkeyword forms is used for control-list specifiers in an I/O statement, the nonkeyword values must appear first. Once a keyword form of a specifier is used, all specifiers to the right must also be keyword forms.

The following sections describe the control-list specifiers in detail.

10.2.1.1. Unit Specifier

The unit specifier identifies the I/O unit to be accessed. It takes the following form:

```
[UNIT=]io-unit
```

io-unit

For external files, it identifies a logical unit and is one of the following:

- A scalar integer expression that refers to a specific file, I/O device, or pipe.If necessary, the value is converted to integer data type before use. The integer is in the range 0 through 2**31–1.

  Units 5 and 6 are associated with preconnected units.

- An asterisk (*). This is the default (or implicit) external unit, which is preconnected for formatted sequential access.

For internal files, *io-unit* identifies a scalar or array character variable that is an internal file. An internal file is designated internal storage space (a variable buffer) that is used with formatted (including list-directed) sequential READ and WRITE statements.

The *io-unit* must be specified in a control list. If the keyword UNIT is omitted, the *io-unit* must be first in the control list.

A unit number is assigned either explicitly through an OPEN statement or implicitly by the system. If a READ statement implicitly opens a file, the file's status is STATUS= 'OLD'. If a WRITE statement implicitly opens a file, the file's status is STATUS= 'NEW'.

If the internal file is a *scalar* character variable, the file has only one record; its length is equal to that of the variable.

If the internal file is an *array* character variable, the file has a record for each element in the array; each record's length is equal to one array element.

An internal file can be read only if the variable has been defined and a value assigned to each record in the file. If the variable representing the internal file is a pointer, it must be associated; if the variable is an allocatable array, it must be currently allocated.

Before data transfer, an internal file is always positioned at the beginning of the first character of the first record.

For More Information:

- On the OPEN statement, see Section 12.6 for details.
- On implicit logical assignments, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On preconnected units, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On using internal files, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.2.1.2. Format Specifier

The format specifier indicates the format to use for data editing. It takes the following form:

```
[FMT=]format
```

`format`

Is one of the following:

- The statement label of a FORMAT statement

  The FORMAT statement must be in the same scoping unit as the data transfer statement.

- An asterisk (*), indicating list-directed formatting

- A scalar default integer variable that has been assigned the label of a FORMAT statement (through an ASSIGN statement)

  The FORMAT statement must be in the same scoping unit as the data transfer statement.

- A character expression (which can be an array or character constant) containing the run-time format

  A default character expression must evaluate to a valid format specification. If the expression is an array, it is treated as if all the elements of the array were specified in array element order and were concatenated.

- The name of a numeric array (or array element) containing the format

If the keyword FMT is omitted, the format specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a format specifier appears in a control list, a namelist group specifier must not appear.

For More Information:

- On FORMAT statements, see Section 11.2.

- On the interaction between FORMAT statements and I/O lists, see Section 11.9.

- On list-directed input, see Section 10.3.1.2; output, see Section 10.5.1.2.

10.2.1.3. Namelist Specifier

The namelist specifier indicates namelist formatting and identifies the namelist group for data transfer. It takes the following form:

```
[NML=]group
```

`group`

Is the name of a namelist group previously declared in a NAMELIST statement.

If the keyword NML is omitted, the namelist specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a namelist specifier appears in a control list, a format specifier must not appear.

For More Information:

On namelist input, see Section 10.3.1.3; output, see Section 10.5.1.3.

10.2.1.4. Record Specifier

The record specifier identifies the number of the record for data transfer in a file connected for direct access. It takes the following form:

```
REC=r
```

r

Is a scalar numeric expression indicating the record number. The value of the expression must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file.

If necessary, the value is converted to integer data type before use.

If REC is present, no END specifier, * format specifier, or namelist group name can appear in the same control list.

For More Information:

On an alternate form of a record specifier, see Section B.8.

10.2.1.5. Key-Field-Value Specifier

The key-field-value specifier identifies the **key** field of a record that you want to access in an indexed file. The key-field value is equal to the contents of a key field. The key field can be used to access records in indexed files because it determines their location.

A key field has attributes, such as the number, direction, length, byte offset, and type of the field. The attributes of the key field are specified at file creation. Records in an indexed file have the same attributes for their key fields.

A key-field-value specifier takes the following form:

```
KEY[con]=val
```

con

Is a selection condition keyword specifying how to compare *val* with key-field values. The keyword can be any of the following:

In Ascending-Key Files:	
Keyword	**Meaning**
EQ	The key-field value must be equal to *val*. KEYEQ is the same as specifying KEY without the optional *con*.
GE	The key-field value must be greater than or equal to *val*.
GT	The key-field value must be greater than *val*.
NXT	The key-field value must be the next value of the key equal to or greater than *val*.

NXTNE	The key-field value must be the next value of the key strictly greater than *val*.
In Descending-Key Files:	
Keyword	**Meaning**
EQ	The key-field value must be equal to *val*. KEYEQ is the same as specifying KEY without the optional *con*.
LE	The key-field value must be less than or equal to *val*.
LT	The key-field value must be less than *val*.
NXT	The key-field value must be the next value of the key equal to or less than *val*.
NXTNE	The key-field value must be the next value of the key that is strictly less than *val*.

`val`

Is an integer or character expression. The expression must match the type of key defined for the file. For an integer key, you must pass an integer expression; it cannot contain real or complex data. For a character key, you can pass either a CHARACTER expression or a BYTE array that contains CHARACTER data.

The specifiers KEY, KEYEQ, KEYNXT, and KEYNXTNE are interchangeable between ascending-key files and descending-key files. However, KEYNXT and KEYNXTNE are interpreted differently depending on the direction of the keys in the file, as follows:

Specifier:	**In Ascending-Key Files**	**In Descending-Key Files**
	Is Equivalent to Specifier:	
KEYNXT	KEYGE	KEYLE
KEYNXTNE	KEYGT	KEYLT

The specifiers KEYGE and KEYGT can only be used with ascending-key files, while the specifiers KEYLE and KEYLT can only be used with descending-key files. Any other use of these key specifiers causes a run-time error to occur.

When a program must be able to use either ascending-key or descending-key files, you should use KEYNXT and KEYNXTNE.

The Selection Process

To select key-field integer values, the process compares values using the signed integers themselves.

To select key-field character values, the process compares values by using the ASCII collating sequence. The comparative length of *val* and a key-field value, as well as any specified selection condition, determine the kind of selection that occurs. The selection can be exact, generic, or approximate-generic, as follows:

- Exact selections occur when the expression in *val* is equal in length to the expression in the key field of the currently accessed record, and the *con* keyword specifies a unique selection condition.
- Generic selections occur when the expression in *val* is shorter than the expression in the key field of the currently accessed record, and the *con* keyword specifies a unique selection condition.

  The process compares all the characters in *val*, from left to right, with the same amount of characters in the key field (also from left to right). Remaining key-field characters are ignored.

For example, consider that a record's key field is 10 characters long and the following statement is entered:

```
READ (3, KEYEQ = 'ABCD')
```

In this case, the process can select a record with a key-field value 'ABCDEFGHIJ '.

- An approximate-generic selection occurs when the expression in *val* is shorter than the expression in the key field, and the *con* keyword *does not* specify a unique selection condition.

  As with generic selections, the process uses only the leftmost characters in the key field to compare values. It selects the first key field that satisfies the generic selection criterion.

  For example, consider that a record's key field is 5 characters long and the following statement is entered:

  ```
  READ (3, KEYGT = 'ABCD')
  ```

  In this case, the process can select the key-field value 'ABCEx ' (and not the key-field value 'ABCDA ').

If *val* is longer than the key-field value, no selection is made and a run-time error occurs.

10.2.1.6. Key-of-Reference Specifier

The key-of-reference specifier can optionally accompany the key-field-value specifier. The key-of-reference specifier indicates the key-field index that is searched to find the designated key-field value. It takes the following form:

```
KEYID=kn
```

kn

Is an integer expression indicating the key-field index. This expression is called the **key of reference**. Its value must be in the range 0 to 254.

A value of zero indicates the **primary key**, a value of 1 indicates the first **alternate key**, a value of 2 indicates the second alternate key, and so forth.

If no *kn* is indicated, the default number is the last specification given in a keyed I/O statement for that I/O unit.

For More Information:

On the key-field-value specifier, see Section 10.2.1.5.

10.2.1.7. I/O Status Specifier

The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

```
IOSTAT=i-var
```

i-var

Is a scalar integer variable. When a data transfer statement is executed, *i-var* is set to one of the following values:

A positive integer	Indicating an error condition occurred.
A negative integer	Indicating an end-of-file or end-of-record condition occurred. The negative integers differ depending on which condition occurred.
Zero	Indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement, or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential READ statement; an end-of-record condition occurs only during execution of a nonadvancing READ statement.

Secondary operating system messages do not display when IOSTAT is specified. To display these messages, remove IOSTAT or use a platform-specific method such as a condition handler.

For More Information:

- On the error numbers returned by IOSTAT, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On condition handlers, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.2.1.8. Branch Specifiers

A branch specifier identifies a branch target statement that receives control if an error, end-of-file, or end-of-record condition occurs. There are three branch specifiers, taking the following forms:

```
ERR=label
END=label
EOR=label
```

`label`

Is the label of the branch target statement that receives control when the specified condition occurs.

The branch target statement must be in the same scoping unit as the data transfer statement.

The following rules apply to these specifiers:

- ERR

  The error specifier can appear in a sequential access READ or WRITE statement, a direct-access READ statement, an indexed READ statement, or a REWRITE statement.

  If an error condition occurs, the position of the file is indeterminate, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a positive integer value. If SIZE was specified (in a nonadvancing READ statement), the SIZE variable becomes defined as an integer value. If an ERR=label was specified, execution continues with the labeled statement.

- END

The end-of-file specifier can appear only in a sequential access READ statement.

An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-of-file record produced by the ENDFILE statement is encountered. End-of-file conditions do not occur in indexed or direct-access READ statements.

If an end-of-file condition occurs, the file is positioned after the end-of-file record, and execution of the statement terminates.

If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If an END=label was specified, execution continues with the labeled statement.

- EOR

  The end-of-record specifier can appear only in a formatted, sequential access READ statement that has the specifier ADVANCE= 'NO' (nonadvancing input).

  An end-of-record condition occurs when a nonadvancing READ statement tries to transfer data from a position after the end of a record.

  If an end-of-record condition occurs, the file is positioned after the current record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If PAD= 'YES' was specified for file connection, the record is padded with blanks (as necessary) to satisfy the input item list and the corresponding data edit descriptor. If SIZE was specified, the SIZE variable becomes defined as an integer value. If an EOR=label was specified, execution continues with the labeled statement.

If one of the conditions occurs, no branch specifier appears in the control list, but an IOSTAT specifier appears, execution continues with the statement following the I/O statement. If neither a branch specifier nor an IOSTAT specifier appears, the program terminates.

For More Information:

- On branch target statements, see Section 7.2.
- On the IOSTAT specifier, see Section 10.2.1.7.
- On error processing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.2.1.9. Advance Specifier

The advance specifier determines whether nonadvancing I/O occurs for a data transfer statement. It takes the following form:

```
ADVANCE=c-expr
```

c-expr

Is a scalar character expression that evaluates to 'YES' for advancing I/O or 'NO' for nonadvancing I/O. The default value is 'YES'.

Trailing blanks in the expression are ignored.

The ADVANCE specifier can appear only in a formatted, sequential data transfer statement that specifies an external unit. It must not be specified for list-directed or namelist data transfer.

Advancing I/O always positions a file at the end of a record, unless an error condition occurs. Nonadvancing I/O can position a file at a character position within the current record.

For More Information:

On advancing and nonadvancing I/O, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.2.1.10. Character Count Specifier

The character count specifier defines a variable to contain the count of how many characters are read when a nonadvancing READ statement terminates. It takes the following form:

```
SIZE=i-var
```

i-var

Is a scalar integer variable.

If PAD= 'YES' was specified for file connection, blanks inserted as padding are not counted.

The SIZE specifier can appear only in a formatted, sequential READ statement that has the specifier ADVANCE= 'NO' (nonadvancing input). It must not be specified for list-directed or namelist data transfer.

10.2.2. I/O Lists

In a data transfer statement, the I/O list specifies the entities whose values will be transferred. The I/O list is either an implied-do list or a simple list of variables (except for assumed-size arrays).

In input statements, the I/O list cannot contain constants and expressions because these do not specify named memory locations that can be referenced later in the program.

However, constants and expressions can appear in the I/O lists for output statements because the compiler can use temporary memory locations to hold these values during the execution of the I/O statement.

If an input item is a pointer, it must be currently associated with a definable target; data is transferred from the file to the associated target. If an output item is a pointer, it must be currently associated with a target; data is transferred from the target to the file.

If an input or output item is an array, it is treated as if the elements (if any) were specified in array element order. For example, if ARRAY_A is an array of shape (2,1), the following input statements are equivalent:

```
READ *, ARRAY_A
READ *, ARRAY_A(1,1), ARRAY_A(2,1)
```

However, no element of that array can affect the value of any expression in the input list, nor can any element appear more than once in an input list. For example, the following input statements are invalid:

```
INTEGER  B(50)
...
```

```
READ *, B(B)
READ *, B(B(1):B(10))
```

If an input or output item is an allocatable array, it must be currently allocated.

If an input or output item is a derived type, the following rules apply:

- Any derived-type component must be in the scoping unit containing the I/O statement.
- The derived type must not have a pointer component.
- In a formatted I/O statement, a derived type is treated as if all of the components of the structure were specified in the same order as in the derived-type definition.
- In an unformatted I/O statement, a derived type is treated as a single object.

The following sections describe simple list items in I/O lists, and implied-do lists in I/O lists.

10.2.2.1. Simple List Items in I/O Lists

In a data transfer statement, a simple list of items takes the following form:

```
item [,item]...
```

item

Is one of the following:

- For input statements: a variable name

  The variable must not be an assumed-size array, unless one of the following appears in the last dimension: a subscript, a vector subscript, or a section subscript specifying an upper bound.

- For output statements: a variable name, expression, or constant

  Any expression must not attempt further I/O operations on the same logical unit. For example, it must not refer to a function subprogram that performs I/O on the same logical unit.

The data transfer statement assigns values to (or transfers values from) the list items in the order in which the items appear, from left to right.

When multiple array names are used in the I/O list of an unformatted input or output statement, only one record is read or written, regardless of how many array name references appear in the list.

Examples

The following example shows a simple I/O list:

```
WRITE (6,10) J, K(3), 4, (L+4)/2, N
```

When you use an array name reference in an I/O list, an input statement reads enough data to fill every item of the array. An output statement writes all of the values in the array.

Data transfer begins with the initial item of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. The following statement defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY appears with no subscripts in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

An input record contains the following values:

```
1,3,721.73
```

The following example shows how variables in the I/O list can be used in array subscripts later in the list:

```
DIMENSION ARRAY(3,3)
...
READ (1,30) J, K, ARRAY(J,K)
```

When the READ statement is executed, the first input value is assigned to J and the second to K, establishing the subscript values for ARRAY(J,K). The value 721.73 is then assigned to ARRAY(1,3). Note that the variables must appear before their use as array subscripts.

Consider the following derived-type definition and structure declaration:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
...
TYPE(EMPLOYEE) :: CONTRACT   ! A structure of type EMPLOYEE
```

The following statements are equivalent:

```
READ *, CONTRACT

READ *, CONTRACT%ID, CONTRACT%NAME
```

For More Information:

On the general rules for I/O lists, see Section 10.2.2.

10.2.2.2. Implied-Do Lists in I/O Lists

In a data transfer statement, an implied-do list acts as though it were a part of an I/O statement within a DO loop. It takes the following form:

```
(list, do-var = expr1, expr2 [,expr3])
```

`list`

Is a list of variables, expressions, or constants (see Section 10.2.2.1).

`do-var`

Is the name of a scalar integer or real variable. The variable must not be one of the input items in *list*.

`expr`

Are scalar numeric expressions of type integer or real. They do not all have to be the same type, or the same type as the DO variable.

The implied-do loop is initiated, executed, and terminated in the same way as a DO construct.

The *list* is the range of the implied-do loop. Items in that list can refer to *do-var*, but they must not change the value of *do-var*.

Two nested implied-do lists must not have the same (or an associated) DO variable.

Use an implied-do list to do the following:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array items in a sequence different from the order of subscript progression

If the I/O statement containing an implied-do list terminates abnormally (with an END, EOR, or ERR branch or with an IOSTAT value other than zero), the DO variable becomes undefined.

Examples

The following two output statements are equivalent:

```
WRITE (3,200) (A,B,C, I=1,3)            ! An implied-do list

WRITE (3,200) A,B,C,A,B,C,A,B,C         ! A simple item list
```

The following example shows nested implied-do lists. Execution of the innermost list is repeated most often:

```
WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
```

The inner DO loop is executed 10 times for each iteration of the outer loop; the second subscript (L) advances from 1 through 10 for each increment of the first subscript (K). This is the reverse of the normal array element order. Note that K is incremented by 2, so only the odd-numbered rows of the array are output.

In the following example, the entire list of the implied-do list (P(1), Q(1,1), Q(1,2)...,Q(1,10)) are read before I is incremented to 2:

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

The following example uses fixed subscripts and subscripts that vary according to the implied-do list:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

Input values are assigned to BOX(1,1) through BOX(1,10), but other elements of the array are not affected.

The following example shows how a DO variable can be output directly:

```
WRITE (6,1111) (I, I=1,20)
```

Integers 1 through 20 are written.

For More Information:

- On the general rules for I/O lists, see Section 10.2.2.
- On DO constructs, see Section 7.6.

10.3. READ Statements

The READ statement is a data transfer input statement. Data can be input from external sequential, keyed-access or direct-access records, or from internal records.

10.3.1. Forms for Sequential READ Statements

Sequential READ statements transfer input data from external sequential-access records. The statements can be formatted with format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

Sequential READ statements take one of the following forms:

```
READ (eunit, format [,advance] [,size] [,iostat] [,err] [,end] [,eor]) [io-
list]
READ form [,io-list]

READ (eunit, * [,iostat] [,err] [,end]) [io-list]
READ * [,io-list]

READ (eunit, nml-group [,iostat] [,err] [,end])
READ nml

READ (eunit [,iostat] [,err] [,end]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format).

`advance`

Is an advance specifier (ADVANCE=c-expr). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.

`size`

Is a character count specifier (SIZE=i-var). It can only be specified for nonadvancing READ statements.

`iostat`

Is a status specifier (IOSTAT=i-var).

`err, end, eor`

Are branch specifiers if an error (ERR=label), end-of-file (END=label), or end-of-record (EOR=label) condition occurs.

EOR can only be specified for nonadvancing READ statements.

`io-list`

Is an I/O list.

`form`

Is the nonkeyword form of a format specifier (no FMT=).

*****

Is the format specifier indicating list-directed formatting.

nml-group

Is a namelist specifier ([NML=]group) indicating namelist formatting.

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On advancing I/O, see Section 10.2.1.9 and the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On file sharing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.3.1.1. Rules for Formatted Sequential READ Statements

Formatted, sequential READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is less than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is greater than the number of fields in an input record, the input record is padded with blanks. However, if PAD= 'NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential READ statements:

```
READ (*, '(B)', ADVANCE='NO') C

READ (FMT="(E2.4)", UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

10.3.1.2. Rules for List-Directed Sequential READ Statements

List-directed, sequential READ statements translate data from character to binary form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then assigned to the entities in the I/O list in the order in which they appear, from left to right.

When a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

List-Directed Records

A list-directed external record consists of a sequence of values and value separators. A value can be any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 4.2).

  - A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.

  - A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  A character string does not need delimiting apostrophes or quotation marks if the corresponding I/O list item is of type default character, and the following is true:

  - The character string does not contain a blank, comma (,), or slash (/).

  - The character string is not continued across a record boundary.

  - The first nonblank character in the string is not an apostrophe or a quotation mark.

  - The leading character is not a string of digits followed by an asterisk.

  A nondelimited character string is terminated by the first blank, comma, slash, or end-of-record encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.

- A null value

A null value is specified by two consecutive value separators (such as ,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Examples

Suppose the following statements are specified:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
```

Then suppose the following external record is read:

```
4 6.3 (3.4,4.2), (3, 2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

The following values are assigned to the I/O list items:

I/O List Item	Value Assigned
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
J	Unchanged
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI 'JK

A	Unchanged
B	Unchanged

For More Information:

- On the literal constant forms of intrinsic data types, see Section 3.2.
- On list-directed output, see Section 10.5.1.2.
- On the general rules for formatted, sequential READ statements, see Section 10.3.1.1.

10.3.1.3. Rules for Namelist Sequential READ Statements

Namelist, sequential READ statements translate data from external to internal form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is assigned to the specified objects in the namelist group in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Namelist Records

A namelist external record takes the following form:

```
&group-name object = value [,object = value].../
```

`group-name`

Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit. The name cannot contain embedded blanks and must be contained within a single record.

`object`

Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks except within the parentheses of a subscript or substring specifier. Each object must be contained in a single record.

`value`

Is any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its

corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 4.2).

- A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.
- A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

A character string does not need delimiting apostrophes or quotation marks if the corresponding NAMELIST item is of type default character, and the following is true:

- The character string does not contain a blank, comma (,), slash (/), exclamation (!), ampersand (&), dollar sign ($), left parenthesis, equal sign (=), percent sign (%), or period (.).
- The character string is not continued across a record boundary.
- The first nonblank character in the string is not an apostrophe or a quotation mark.
- The leading character is not a string of digits followed by an asterisk.

A nondelimited character string is terminated by the first blank, comma, slash, end-of-record, exclamation, ampersand, or dollar sign encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.

If an equal sign, percent sign, or period is encountered while scanning for a nondelimited character string, the string is treated as a variable name (or part of one) and not as a nondelimited character string.

- A null value

  A null value is specified by two consecutive value separators (such as ,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value).

  A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

Blanks can precede or follow the beginning ampersand (&), follow the group name, precede or follow the equal sign, or precede the terminating slash.

Comments (beginning with ! only) can appear anywhere in namelist input. The comment extends to the end of the source line.

If an entity appears more than once within the input record for a namelist data transfer, the last value is the one that is used.

If there is more than one *object=value* pair, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Prompting for Namelist Group Information

During execution of a program containing a namelist READ statement, you can specify a question mark character (?) or a question mark character preceded by an equal sign (=?) to get information about the namelist group. The ? or =? must follow one or more blanks.

If specified for a unit capable of both input and output, the ? causes display of the group name and the objects in that group. The =? causes display of the group name, objects within that group, and the current values for those objects (in namelist output form). If specified for another type of unit, the symbols are ignored.

For example, consider the following statements:

```
NAMELIST /NLIST/ A,B,C
REAL A /1.5/
INTEGER B /2/
CHARACTER*5 C /'ABCDE'/

READ (5,NML=NLIST)
WRITE (6,NML=NLIST)
END
```

During execution, if a blank followed by ? is entered on a terminal device, the following values are displayed:

```
 &NLIST
   A
   B
   C
 /
```

If a blank followed by =? is entered, the following values are displayed:

```
 &NLIST
   A       =   1.500000    ,
   B       =           2,
   C       = ABCDE
 /
```

Examples

Suppose the following statements are specified:

```
NAMELIST /CONTROL/ TITLE, RESET, START, STOP, INTERVAL
CHARACTER*10 TITLE
REAL(KIND=8) START, STOP
LOGICAL(KIND=4) RESET
INTEGER(KIND=4) INTERVAL
```

```
READ (UNIT=1, NML=CONTROL)
```

The NAMELIST statement associates the group name CONTROL with a list of five objects. The corresponding READ statement reads the following input data from unit 1:

```
&CONTROL
   TITLE='TESTT002AA',
   INTERVAL=1,
   RESET=.TRUE.,
   START=10.2,
   STOP =14.5
/
```

The following values are assigned to objects in group CONTROL:

Namelist Object	Value Assigned
TITLE	TESTT002AA
RESET	T
START	10.2
STOP	14.5
INTERVAL	1

It is not necessary to assign values to all of the objects declared in the corresponding NAMELIST group. If a namelist object does not appear in the input statement, its value (if any) is unchanged.

Similarly, when character substrings and array elements are specified, only the values of the specified variable substrings and array elements are changed. For example, suppose the following input is read:

```
&CONTROL TITLE(9:10)='BB' /
```

The new value for TITLE is TESTT002BB; only the last two characters in the variable change.

The following example shows an array as an object:

```
DIMENSION ARRAY_A(20)
NAMELIST /ELEM/ ARRAY_A
READ (UNIT=1,NML=ELEM)
```

Suppose the following input is read:

```
&ELEM
ARRAY_A=1.1, 1.2, , 1.4
/
```

The following values are assigned to the ARRAY_A elements:

Array Element	Value Assigned
ARRAY_A(1)	1.1
ARRAY_A(2)	1.2
ARRAY_A(3)	Unchanged
ARRAY_A(4)	1.4
ARRAY_A(5)...ARRAY(20)	Unchanged

When a list of values is assigned to an array element, the assignment begins with the specified array element, rather than with the first element of the array. For example, suppose the following input is read:

```
&ELEM
ARRAY_A(3)=34.54, 45.34, 87.63, 3*20.00
/
```

New values are assigned only to array ARRAY_A elements 3 through 8. The other element values are unchanged.

Nondelimited character strings that are written out by using a NAMELIST write may not be read in as expected by a corresponding NAMELIST read. Consider the following:

```
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/'AAA', 'BBB', 'CCC', 'DDD'/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
WRITE (1, NML=TEST)
END
```

The output file NMLTEST.DAT will contain:

```
&TEST
CHARR   = AAABBBCCCDDD
/
```

If an attempt is then made to read the data in NMLTEST.DAT with a NAMELIST read using nondelimited character strings, as follows:

```
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/4*'   '/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
READ (1, NML=TEST)
PRINT *, 'CHARR read in >', CHARR(1),'<  >',CHARR(2),'<  >',
1       CHARR(3), '<  >', CHARR(4), '<'
END
```

The result is the following:

```
CHARR read in >AAA<  <   >   <  >   <  >   <
```

For More Information:

- On the NAMELIST statement, in general, and rules for objects in a namelist group, see Section 5.12.
- On an alternative form for namelist external records, see Section B.10.
- On namelist output, see Section 10.5.1.3.
- On the general rules for formatted, sequential READ statements, see Section 10.3.1.1.

10.3.1.4. Rules for Unformatted Sequential READ Statements

Unformatted, sequential READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, sequential READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is *greater* than the number of fields in an input record, an error occurs.

If a statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows an unformatted, sequential READ statement:

```
READ (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

10.3.2. Forms for Direct-Access READ Statements

Direct-access READ statements transfer input data from external records with direct access. (The attributes of a direct-access file are established by the OPEN statement).

A direct-access READ statement can be formatted or unformatted, and takes one of the following forms:

```
READ (eunit, format, rec [,iostat] [,err]) [io-list]

READ (eunit, rec [,iostat] [,err]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format). It must not be an asterisk (*).

`rec`

Is a record specifier (REC=r).

`iostat`

Is a status specifier (IOSTAT=i-var).

`err`

Is a branch specifier (ERR=label) if an error condition occurs.

io-list

Is an I/O list.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On file sharing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.3.2.1. Rules for Formatted Direct-Access READ Statements

Formatted, direct-access READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD= 'NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is read by that input statement.

Examples

The following example shows a formatted, direct-access READ statement:

```
READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

10.3.2.2. Rules for Unformatted Direct-Access READ Statements

Unformatted, direct-access READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, direct-access READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is less than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is greater than the number of fields in an input record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access READ statements:

```
READ (1, REC=10) LIST(1), LIST(8)
READ (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

10.3.3. Forms for Indexed READ Statements

Indexed READ statements transfer input data from external records that have **keyed access**.

In an indexed file, a series of records can be read in key value sequence by using an indexed READ statement and sequential READ statements. The first record in the sequence is read by using the indexed statement, the rest are read by using the sequential READ statements.

An indexed READ statement can be formatted or unformatted, and takes one of the following forms:

```
READ (eunit, format, key [,keyid] [,iostat] [,err]) [io-list]

READ (eunit, key [,keyid] [,iostat] [,err]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format).

`key`

Is a key specifier (KEY[con]=value).

`keyid`

Is a key-of-reference specifier (KEYID=kn).

`iostat`

Is a status specifier (IOSTAT=i-var).

`err`

Is a branch specifier (ERR=label) if an error condition occurs.

`io-list`

Is an I/O list.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.

10.3.3.1. Rules for Formatted Indexed READ Statements

Formatted, indexed READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

If the I/O list and format specifications indicate that additional records are to be read, the statement reads the additional records sequentially by using the current key-of-reference value.

If KEYID is omitted, the key-of-reference value is the same as the most recent specification. If KEYID is omitted from the first indexed READ statement, the key of reference is the primary key.

If the specified key value is shorter than the key field referenced, the key value is matched against the leftmost characters of the appropriate key field until a match is found. The record supplying the match is then read. If the key value is longer than the key field referenced, an error occurs.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

Suppose the following statement is specified:

```
READ (3, KAT(25), KEY='ABCD') A,B,C,D
```

The READ statement retrieves a record with a key value of 'ABCD' in the primary key from the file connected to I/O unit 3. It then uses the format contained in the array item KAT(25) to read the first four fields from the record into variables A,B,C, and D.

10.3.3.2. Rules for Unformatted Indexed READ Statements

Unformatted, indexed READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

If the number of I/O list items is less than the number of fields in the record being read, the unused fields in the record are discarded. If the number of I/O list items is greater than the number of fields, an error occurs.

If a specified key value is shorter than the key field referenced, the key value is matched against the leftmost characters of the appropriate key field until a match is found. The record supplying the match is then read. If the specified key value is longer than the key field referenced, an error occurs.

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Examples

Suppose the following statements are specified:

```
      OPEN (UNIT=3, STATUS='OLD',
1     ACCESS='KEYED', ORGANIZATION='INDEXED',
```

```
2     FORM='UNFORMATTED',
3     KEY=(1:5,30:37,18:23))
      READ (3,KEY='SMITH') ALPHA, BETA
```

The READ statement reads from the file connected to I/O unit 3 and retrieves the record with the value 'SMITH' in the primary key field (bytes 1 through 5). The first two fields of the record retrieved are placed in variables ALPHA and BETA, respectively.

Suppose the following statement is specified:

```
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
```

In this case, the READ statement retrieves the first record having a value equal to or greater than 'XYZDEF' in the second alternate key field (bytes 18 through 23). The first field of that record is placed in variable IKEY.

10.3.4. Forms and Rules for Internal READ Statements

Internal READ statements transfer input data from an internal file.

An internal READ statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is not permitted.

An internal READ statement takes the following form:

```
READ (iunit, format [,iostat] [,err] [,end]) [io-list]
```

iunit

Is an internal unit specifier ([UNIT=]io-unit). It must be a character variable. It must not be an array section with a vector subscript.

format

Is a format specifier ([FMT=]format). An asterisk (*) indicates list-directed formatting.

iostat

Is a status specifier (IOSTAT=i-var).

err, end

Are branch specifiers if an error (ERR=label) or end-of-file (END=label) condition occurs.

io-list

Is an I/O list.

Formatted, internal READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

This form of READ statement behaves as if the format begins with a BN edit descriptor. (You can override this behavior by explicitly specifying the BZ edit descriptor).

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

Before data transfer occurs, the file is positioned at the beginning of the first record. This record becomes the current record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD= 'NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains.

In list-directed formatting, character strings have no delimiters.

Examples

An internal read can be used to convert character data to numeric values.

The following program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal READ statements to make appropriate conversions from character string representations to binary.

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*(*) AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',        &
           ZFMT= '(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE = RECORD(1:1)
IF (TYPE .EQ. 'D') THEN
    READ (RECORD(2:MIN(ILEN, 11)), IFMT) IVAL
ELSE IF (TYPE .EQ. 'O') THEN
    READ (RECORD(2:MIN(ILEN, 12)), OFMT) IVAL
ELSE IF (TYPE .EQ. 'X') THEN
    READ (RECORD(2:MIN(ILEN, 9)),ZFMT) IVAL
ELSE
    PRINT *, 'ERROR'
END IF
END
```

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On list-directed input, see Section 10.3.1.2.
- On using internal files, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.4. ACCEPT Statement

The ACCEPT statement is a data transfer input statement. This statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units.

An ACCEPT statement takes one of the following forms:

```
ACCEPT form [,io-list]

ACCEPT * [,io-list]

ACCEPT nml
```

form

Is the nonkeyword form of a format specifier (no FMT=).

io-list

Is an I/O list.

*****

Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Examples

In the following example, character data is read from the implicit unit and binary values are assigned to each of the five elements of array CHARAR:

```
      CHARACTER*10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

For More Information:

- On formatted, sequential READ statements, see Section 10.3.1.1.
- On formatted data and data transfers, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On list-directed input, see Section 10.3.1.2.
- On namelist input, see Section 10.3.1.3.
- On I/O lists, see Section 10.2.2.

10.5. WRITE Statements

The WRITE statement is a data transfer output statement. Data can be output to external sequential, keyed-access, or direct-access records, or to internal records.

10.5.1. Forms for Sequential WRITE Statements

Sequential WRITE statements transfer output data to external sequential access records. The statements can be formatted by using format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential WRITE statement takes one of the following forms:

```
WRITE (eunit, format [,advance] [,iostat] [,err]) [io-list]

WRITE (eunit, * [,iostat] [,err]) [io-list]

WRITE (eunit, nml-group [,iostat] [,err])

WRITE (eunit [,iostat] [,err]) [io-list]
```

eunit

Is an external unit specifier ([UNIT=]io-unit).

format

Is a format specifier ([FMT=]format).

advance

Is an advance specifier (ADVANCE=c-expr). If the value of *c-expr* is 'YES', the statement uses advancing output; if the value is 'NO', the statement uses nonadvancing output. The default value is 'YES'.

iostat

Is a status specifier (IOSTAT=i-var).

err

Is a branch specifier (ERR=label) if an error condition occurs.

io-list

Is an I/O list.

*****

Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml-group

Is a namelist specifier ([NML=]group) indicating namelist formatting.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On advancing I/O, see Section 10.2.1.9 and the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.5.1.1. Rules for Formatted Sequential WRITE Statements

Formatted, sequential WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for sequential access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

The output list and format specification must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement.)

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential WRITE statements:

```
WRITE (UNIT=8, FMT='(B)', ADVANCE='NO') C

WRITE (*, "(F6.5)", ERR=25, IOSTAT=IO_STATUS) A, B, C
```

10.5.1.2. Rules for List-Directed Sequential WRITE Statements

List-directed, sequential WRITE statements transfer data from binary to character form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

Table 10.1 shows the default output formats for each intrinsic data type.

Table 10.1. Default Formats for List-Directed Output

Data Type	Output Format
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8)	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8)	I22
REAL(4)	1PG15.7E2
REAL(8) T_floating	1PG24.15E3
REAL(8) D_floating	1PG24.16E2
REAL(8) G_floating	1PG24.15E3
REAL(16)	1PG43.33E4
COMPLEX(4)	' (',1PG14.7E2, ', ',1PG14.7E2, ') '
COMPLEX(8) T_floating	' (',1PG23.15E3, ', ',1PG23.15E3, ') '

Data Type	Output Format
COMPLEX(8) D_floating	' (',1PG23.16E2, ', ',1PG23.16E2, ') '
COMPLEX(8) G_floating	' (',1PG23.15E3, ', ',1PG23.15E3, ') '
COMPLEX(16)	' (',1PG42.33E4, ', ',1PG42.33E4, ') '
CHARACTER	A *w*[1]

[1]Where *w* is the length of the character expression.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM= 'QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM= 'APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. In the case of complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record.

Each output record begins with a blank character for carriage control, except for literal character constants that are continued from the previous record.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

Examples

Suppose the following statements are specified:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE (1,*) 'ARRAY VALUES FOLLOW'
WRITE (1,*) A,4
```

The following records are then written to external unit 1:

```
ARRAY VALUES FOLLOW
   3.400000      3.400000      3.400000      3.400000      4
```

For More Information:

- On list-directed input, see Section 10.3.1.2.
- On general rules for formatted, sequential WRITE statements, see Section 10.5.1.1.

10.5.1.3. Rules for Namelist Sequential WRITE Statements

Namelist, sequential WRITE statements translate data from internal to external form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM= 'QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM= 'APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. In the case of complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record.

Each output record begins with a blank character for carriage control, except for literal character constants that are continued from the previous record.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Examples

Consider the following statements:

```
CHARACTER*19 NAME(2)/2*' '/
REAL PITCH, ROLL, YAW, POSITION(3)
LOGICAL DIAGNOSTICS
INTEGER ITERATIONS
NAMELIST /PARAM/ NAME, PITCH, ROLL, YAW, POSITION,        &
         DIAGNOSTICS, ITERATIONS
...
READ (UNIT=1,NML=PARAM)
WRITE (UNIT=2,NML=PARAM)
```

Suppose the following input is read:

```
&PARAM
    NAME(2)(10:)='HEISENBERG',
    PITCH=5.0, YAW=0.0, ROLL=5.0,
    DIAGNOSTICS=.TRUE.
    ITERATIONS=10
/
```

The following is then written to the file connected to unit 2:

```
&PARAM
NAME   = '                          ', '        HEISENBERG',
PITCH  =    5.000000    ,
ROLL   =    5.000000    ,
YAW    =   0.0000000E+00,
POSITION        = 3*0.0000000E+00,
DIAGNOSTICS     = T,
ITERATIONS      =               10
/
```

Note that character values are not enclosed in apostrophes unless the output file is opened with DELIM= 'APOSTROPHE'. The value of POSITION is not defined in the namelist input, so the current value of POSITION is written.

For More Information:

- On namelist input, see Section 10.3.1.3.
- On general rules for formatted, sequential WRITE statements, see Section 10.5.1.1.

10.5.1.4. Rules for Unformatted Sequential WRITE Statements

Unformatted, sequential WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

This form of WRITE statement writes exactly one record. If there is no I/O item list, the statement writes one null record.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows an unformatted, sequential WRITE statement:

```
WRITE (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

10.5.2. Forms for Direct-Access WRITE Statements

Direct-access WRITE statements transfer output data to external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access WRITE statement can be formatted or unformatted, and takes one of the following forms:

```
WRITE (eunit, format, rec [,iostat] [,err]) [io-list]

WRITE (eunit, rec [,iostat] [,err]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format). It must not be an asterisk (*).

`rec`

Is a record specifier (REC=r).

`iostat`

Is a status specifier (IOSTAT=i-var).

`err`

Is a branch specifier (ERR=label) if an error condition occurs.

`io-list`

Is an I/O list.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.

10.5.2.1. Rules for Formatted Direct-Access WRITE Statements

Formatted, direct-access WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for direct access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the format specification specifies another record, the record number is increased by 1 as each subsequent record is written by that output statement.

Examples

The following example shows a formatted, direct-access WRITE statement:

```
WRITE (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

10.5.2.2. Rules for Unformatted Direct-Access WRITE Statements

Unformatted, direct-access WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access WRITE statements:

```
WRITE (1, REC=10) LIST(1), LIST(8)
WRITE (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

10.5.3. Forms for Indexed WRITE Statements

Indexed WRITE statements transfer output data to external records that have keyed access. (The OPEN statement establishes the attributes of an indexed file).

Indexed WRITE statements always write a new record. You should use the REWRITE statement to update an existing record.

The syntax of an indexed WRITE statement is similar to a sequential WRITE statement, but an indexed WRITE statement refers to an I/O unit connected to an indexed file, whereas the sequential WRITE statement refers to an I/O unit connected to a sequential file.

An indexed WRITE statement can be formatted or unformatted, and takes one of the following forms:

```
WRITE (eunit, format, [,iostat] [,err]) [io-list]

WRITE (eunit, [,iostat] [,err]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format).

`iostat`

Is a status specifier (IOSTAT=i-var).

`err`

Is a branch specifier (ERR=label) if an error condition occurs.

`io-list`

Is an I/O list.

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.

- On the REWRITE statement, see Section 10.7.

10.5.3.1. Rules for Formatted Indexed WRITE Statements

Formatted, indexed WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for keyed access.

No key parameters are required in the list of control parameters, because all necessary key information is contained in the output record.

When you use a formatted indexed WRITE statement to write an INTEGER key, the key is translated from internal binary form to external character form. A subsequent attempt to read the record by using an integer key may not match the key field in the record.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

Consider the following example (which assumes that the first 10 bytes of a record are a character key):

```
      WRITE (4,100) KEYVAL, (RDATA(I), I=1, 20)
100   FORMAT (A10, 20F15.7)
```

The WRITE statement writes the translated values of each of the 20 elements of the array RDATA to a new formatted record in the indexed file connected to I/O unit 4. KEYVAL is the key by which the record is accessed.

10.5.3.2. Rules for Unformatted Indexed WRITE Statements

Unformatted, indexed WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record.

No key parameters are required in the list of control parameters, because all necessary key information is contained in the output record.

If the values specified by the I/O list do not fill a fixed-length record being written, the unused portion of the record is filled with zeros. If the values specified do not fit in the record, an error occurs.

Since derived data types of sequence type usually have a fixed record format, you can write to indexed files by using a sequence derived-type structure that models the file's record format. This lets you perform the I/O operation with a single derived-type variable instead of a potentially long I/O list. Nonsequence derived types should not be used for this purpose.

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Examples

The following example shows an unformatted, indexed WRITE statement:

```
WRITE (UNIT=8, IOSTAT=IO_STATUS) A, B, C
```

10.5.4. Forms and Rules for Internal WRITE Statements

Internal WRITE statements transfer output data to an internal file.

An internal WRITE statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is not permitted.

An internal WRITE statement takes the following form:

```
WRITE (iunit, format [,iostat] [,err]) [io-list]
```

iunit

Is an internal unit specifier ([UNIT=]io-unit). It must be a default character variable. It must not be an array section with a vector subscript.

format

Is a format specifier ([FMT=]format). An asterisk (*) indicates list-directed formatting.

iostat

Is a status specifier (IOSTAT=i-var).

err

Is a branch specifier (ERR=label) if an error condition occurs.

io-list

Is an I/O list.

Formatted, internal WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an internal file.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the number of characters written in a record is less than the length of the record, the rest of the record is filled with blanks. The number of characters to be written must not exceed the length of the record.

Character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

Examples

The following example shows an internal WRITE statement:

```
INTEGER J, K, STAT_VALUE
CHARACTER*50 CHAR_50
...
WRITE (FMT=*, UNIT=CHAR_50, IOSTAT=STAT_VALUE) J, K
```

For More Information:

- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On list-directed output, see Section 10.5.1.2.

- On using internal files, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

10.6. PRINT and TYPE Statements

The PRINT statement is a data transfer output statement. TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.

The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units.

A PRINT statement takes one of the following forms:

```
PRINT form [,io-list]

PRINT * [,io-list]

PRINT nml
```

form

Is the nonkeyword form of a format specifier (no FMT=).

io-list

Is an I/O list.

*****

Is the format specifier indicating list-directed formatting.

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Examples

In the following example, one record (containing four fields of data) is printed to the implicit output device:

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=', A, 'JOB=', A)
```

For More Information:

- On formatted, sequential WRITE statements, see Section 10.5.1.1.
- On formatted data and data transfers, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On list-directed output, see Section 10.5.1.2.
- On namelist output, see Section 10.5.1.3.
- On I/O lists, see Section 10.2.2.

10.7. REWRITE Statement

The REWRITE statement is a data transfer output statement that rewrites the current record.

A REWRITE statement can be formatted or unformatted, and takes one of the following forms:

```
REWRITE (eunit, format [,iostat] [,err]) [io-list]

REWRITE (eunit [,iostat] [,err]) [io-list]
```

`eunit`

Is an external unit specifier ([UNIT=]io-unit).

`format`

Is a format specifier ([FMT=]format).

`iostat`

Is a status specifier (IOSTAT=i-var).

`err`

Is a branch specifier (ERR=label) if an error condition occurs.

`io-list`

Is an I/O list.

In the REWRITE statement, data (translated if formatted; untranslated if unformatted) is written to the current (existing) record in one of the following types of external files:

- In all types of files. In sequential files, the current record and new record must be the same length.

The current record is the last record accessed by a preceding, successful sequential, indexed, or direct-access READ statement.

Between a READ and REWRITE statement, you should not specify any other I/O statement (except INQUIRE) on that logical unit. Execution of any other I/O statement on the logical unit destroys the current-record context and causes the current record to become undefined.

Only one record can be rewritten in a single REWRITE statement operation.

The output list (and format specification, if any) must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement).

If the number of characters specified by the I/O list (and format, if any) do not fill a record, blank characters are added to fill the record.

If the primary key value is changed in a keyed-access file, an error occurs.

Examples

In the following example, the current record (contained in the relative organization file connected to logical unit 3) is updated with the values represented by NAME, AGE, and BIRTH:

```
      REWRITE (3, 10, ERR=99) NAME, AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

For More Information:

- On formatted data and data transfers, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On I/O control-list specifiers, see Section 10.2.1.
- On I/O lists, see Section 10.2.2.
- On the RECL specifier in OPEN statements, see Section 12.6.25 for details.

Chapter 11. I/O Formatting

11.1. Overview

A format appearing in an input or output (I/O) statement specifies the form of data being transferred and the data conversion (editing) required to achieve that form. The format specified can be explicit or implicit.

Explicit format is indicated in a format specification that appears in a FORMAT statement or a character expression (the expression must evaluate to a valid format specification).

The format specification contains edit descriptors, which can be data edit descriptors, control edit descriptors, or string edit descriptors.

Implicit format is determined by the processor and is specified using list-directed or namelist formatting.

List-directed formatting is specified with an asterisk (*); namelist formatting is specified with a namelist group name.

List-directed formatting can be specified for advancing sequential files and internal files. Namelist formatting can be specified only for advancing sequential files.

For More Information:

- On list-directed input, see Section 10.3.1.2; output, see Section 10.5.1.2.
- On namelist input, see Section 10.3.1.3; output, see Section 10.5.1.3.

11.2. Format Specifications

A format specification can appear in a FORMAT statement or character expression. In a FORMAT statement, it is preceded by the keyword FORMAT. A format specification takes the following form:

```
(format-list)
```

`format-list`

Is a list of one or more of the following edit descriptors, separated by commas or slashes (/):

Data edit descriptors:	I, B, O, Z, F, E, EN, ES, D, G, L, and A.
Control edit descriptors:	T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, Δ, \, and Q.
String edit descriptors:	H, 'c', and "c ", where *c* is a character constant.

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Edit descriptors can be nested and a *repeat specification* can precede data edit descriptors, the slash edit descriptor, or a parenthesized list of edit descriptors.

Rules and Behavior

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see Section 12.6.4).

For formatted input, use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a blank, 0, 1, Δ, +, or ASCII NUL. Any other character is treated as a blank.

A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

Table 11.1 summarizes the edit descriptors that can be used in format specifications.

Table 11.1. Summary of Edit Descriptors

Code	Form	Effect	
A	A[w]	Transfers character or Hollerith values.	(Section 11.3.6)
B	Bw[.m]	Transfers binary values.	(Section 11.3.3.2)
BN	BN	Ignores embedded and trailing blanks in a numeric input field.	(Section 11.4.4.1)
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.	(Section 11.4.4.2)
D	Dw.d	Transfers real values with D exponents.	(Section 11.3.4.2)
E	Ew.d[Ee]	Transfers real values with E exponents.	(Section 11.3.4.2)
EN	ENw.d[Ee]	Transfers real values with engineering notation.	(Section 11.3.4.3)
ES	ESw.d[Ee]	Transfers real values with scientific notation.	(Section 11.3.4.4)

Code	Form	Effect	
F	Fw.d	Transfers real values with no exponent.	(Section 11.3.4.1)
G	Gw.d[Ee]	Transfers values of all intrinsic types.	(Section 11.3.4.5)
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record.	(Section 11.5)
I	Iw[.m]	Transfers decimal integer values.	(Section 11.3.3.1)
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.	(Section 11.3.5)
O	Ow[.m]	Transfers octal values.	(Section 11.3.3.3)
P	kP	Interprets certain real numbers with a specified scale factor.	(Section 11.4.5)
Q	Q	Returns the number of characters remaining in an input record.	(Section 11.4.9)
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.	(Section 11.4.3.3)
SP	SP	Writes optional plus sign (+) into numeric output fields.	(Section 11.4.3.1)
SS	SS	Suppresses optional plus sign (+) in numeric output fields.	(Section 11.4.3.2)
T	Tn	Tabs to specified position.	(Section 11.4.2.1)
TL	TLn	Tabs left the specified number of positions.	(Section 11.4.2.2)
TR	TRn	Tabs right the specified number of positions.	(Section 11.4.2.3)
X	nX	Skips the specified number of positions.	(Section 11.4.2.4)
Z	Zw[.m]	Transfers hexadecimal values.	(Section 11.3.3.4)
Δ	Δ	Suppresses trailing carriage return during interactive I/O.	(Section 11.4.8)
:	:	Terminates format control if there are no more items in the I/O list.	(Section 11.4.7)
/	[r]/	Terminates the current record and moves to the next record.	(Section 11.4.6)
\	\	Continues the same record; same as Δ.	(Section 11.4.8)
'c'[1]	'c'	Transfers the character literal constant (between the delimiters) to an output record.	(Section 11.5)

[1]These delimiters can also be quotation marks (").

Character Format Specifications

In data transfer I/O statements, a format specifier ([FMT=]format) can be a character expression that is a character array, character array element, or character constant. This type of format is also called a run-time format because it can be constructed or altered during program execution.

The expression must evaluate to a character string whose leading part is a valid format specification (including the enclosing parentheses).

If the expression is a character array element, the format specification must be contained entirely within that element.

If the expression is a character array, the format specification can continue past the first element into subsequent consecutive elements.

If the expression is a character constant delimited by apostrophes, use two consecutive apostrophes (') to represent an apostrophe character in the format specification; for example:

```
PRINT '("NUM can''t be a real number")'
```

Similarly, if the expression is a character constant delimited by quotation marks, use two consecutive quotation marks (" ") to represent a quotation mark character in the format specification.

To avoid using consecutive apostrophes or quotation marks, you can put the character constant in an I/O list instead of a format specification, as follows:

```
PRINT "(A)", "NUM can't be a real number"
```

The following shows another character format specification:

```
WRITE (6, '(I12, I4, I12)') I, J, K
```

In the following example, the format specification changes with each iteration of the DO loop:

```
SUBROUTINE PRINT(TABLE)
REAL TABLE(10,5)
CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG, FMED, FSML
DATA FORCHR(0),RPAR /'(',')'/
DATA FBIG,FMED,FSML /'F8.2,','F9.4,','F9.6,'/
DO I=1,10
  DO J=1,5
    IF (TABLE(I,J) .GE. 100.) THEN
      FORCHR(J) = FBIG
    ELSE IF (TABLE(I,J) .GT. 0.1) THEN
      FORCHR(J) = FMED
    ELSE
      FORCHR(J) = FSML
    END IF
  END DO
  FORCHR(5)(5:5) = RPAR
  WRITE (6,FORCHR) (TABLE(I,J), J=1,5)
END DO
END
```

The DATA statement assigns a left parenthesis to character array element FORCHR (0), and (for later use) a right parenthesis and three F edit descriptors to character variables.

Next, the proper F edit descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of array TABLE.

A right parenthesis is added to the format specification just before the WRITE statement uses it.

Note

Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a READ statement to read data into the format itself, that data is not copied back into the original array, and the array is unavailable for subsequent use as a run-time format specification.

For More Information:

- On data edit descriptors, see Section 11.3.
- On control edit descriptors, see Section 11.4.
- On character string edit descriptors, see Section 11.5.
- On nested and group repeats, see Section 11.6.
- On printing of formatted records, see Section 11.8.

11.3. Data Edit Descriptors

A data edit descriptor causes the transfer or conversion of data to or from its internal representation.

The part of a record that is input or output and formatted with data edit descriptors (or character string edit descriptors) is called a **field**.

This section describes the forms for data edit descriptors and the individual descriptors, themselves. It also describes general rules for numeric editing and default widths for data edit descriptors.

11.3.1. Forms for Data Edit Descriptors

A data edit descriptor takes one of the following forms:

```
[r]c
[r]cw
[r]cw.m
[r]cw.d
[r]cw.d[Ee]
```

r

Is a repeat specification. The range of *r* is 1 through 2147483647 (2**31–1). If *r* is omitted, it is assumed to be 1.

c

Is one of the following format codes: I, B, O, Z, F, E, EN, ES, D, G, L, or A.

w

Is the total number of digits in the field (the field width). If omitted, the system applies default values (see Section 11.3.7). The range of *w* is 1 through 2147483647 (2**31–1). For I, B, O, Z, and F, the range can start at zero.

m

Is the minimum number of digits that must be in the field (including leading zeros). The range of *m* is 0 through 32767 (2**15–1).

d

Is the number of digits to the right of the decimal point (the significant digits). The range of *d* is 0 through 32767 (2**15–1).

The number of significant digits is affected if a scale factor is specified for the data edit descriptor.

E

Identifies an exponent field.

e

Is the number of digits in the exponent. The range of *e* is 1 through 32767 (2**15–1).

Rules and Behavior

Fortran 95/90 (and the previous standard) allows the field width to be omitted only for the A descriptor. However, VSI Fortran allows the field width to be omitted for any data edit descriptor.

The *r*, *w*, *m*, *d*, and *e* must all be positive, unsigned, integer literal constants; or variable format expressions; no kind parameter can be specified. They must not be named constants.

Actual useful ranges for *r*, *w*, *m*, *d*, and *e* may be constrained by record sizes (RECL) and the file system.

The data edit descriptors have the following specific forms:

Integer:	Iw[.m], Bw[.m], Ow[.m], and Zw[.m]
Real and complex:	Fw.d, Ew.d[Ee], ENw.d[Ee], ESw.d[Ee], Dw.d, and Gw.d[Ee]
Logical:	Lw
Character:	A[w]

The *d* must be specified with F, E, D, and G field descriptors even if *d* is zero. The decimal point is also required. You must specify both *w* and *d*, or omit them both.

A repeat specification can simplify formatting. For example, the following two statements are equivalent:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
20  FORMAT (3E12.4,4I5)
```

For More Information:

- On general rules for numeric editing, see Section 11.3.2.
- On nested and group repeats, see Section 11.6.

11.3.2. General Rules for Numeric Editing

The following rules apply to input and output data for numeric editing (data edit descriptors I, B, O, Z, F, E, EN, ES, D, and G).

Rules for Input Processing

Leading blanks in the external field are ignored. If BLANK= 'NULL' is in effect (or the BN edit descriptor has been specified) embedded and trailing blanks are ignored; otherwise, they are treated as zeros. An all-blank field is treated as a value of zero.

The following table shows how blanks are interpreted by default:

Type of Unit or File	**Default**
An explicitly OPENed unit	BLANK= 'NULL'
An internal file	BLANK= 'NULL'
A preconnected file[1]	BLANK= 'NULL'

[1]For interactive input from preconnected files, you should explicitly specify the BN or BZ edit descriptor to ensure desired behavior.

A minus sign must precede a negative value in an external field; a plus sign is optional before a positive value.

In input records, constants can include any valid kind parameter. Named constants are not permitted.

If the data field in a record contains fewer than *w* characters, an input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data. The comma terminates the data field, and can also be used to designate null (zero-length) fields. For more information, see Section 11.3.8.

Rules for Output Processing

The field width *w* must be large enough to include any leading plus or minus sign, and any decimal point or exponent. For example, the field width for an E data edit descriptor must be large enough to contain the following:

- For positive numbers: *d*+5 or *d*+ *e*+3 characters
- For negative numbers: *d*+6 or *d*+ *e*+4 characters

A positive or zero value (zero is allowed for I, B, O, Z, and F descriptors) can have a plus sign, depending on which sign edit descriptor is in effect. If a value is negative, the leftmost nonblank character is a minus sign.

If the value is smaller than the field width specified, leading blanks are inserted (the value is right-justified). If the value is too large for the field width specified, the entire output field is filled with asterisks (*).

When the value of the field width is zero, the compiler selects the smallest possible positive actual field width that does not result in the field being filled with asterisks.

For More Information:

- On format specifications, in general, see Section 11.2.

- On the form for data edit descriptors, see Section 11.3.1.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

11.3.3. Integer Editing

Integer editing is controlled by the I (decimal), B (binary), O (octal), and Z (hexadecimal) data edit descriptors.

11.3.3.1. I Editing

The I edit descriptor transfers decimal integer values. It takes the following form:

```
Iw[.m]
```

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item must be of type integer or logical

The G edit descriptor can be used to edit integer data; it follows the same rules as I *w*.

Rules for Input Processing

On input, the I data edit descriptor transfers *w* characters from an external field and assigns their integer value to the corresponding I/O list item. The external field data must be an integer constant.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the I edit descriptor:

Format	Input	Value
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	312

Rules for Output Processing

On output, the I data edit descriptor transfers the value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by a sign (a plus sign is optional for positive values, a minus sign is required for negative values), followed by an unsigned integer constant with no leading zeros.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the I edit descriptor:

Format	Value	Output
I3	284	284

Format	Value	Output
I4	-284	-284
I4	0	ΔΔΔ0
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.812	An error; the decimal point is invalid
I4.0	0	ΔΔΔΔ
I4.2	1	ΔΔ01
I4.4	1	0001

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.3.2. B Editing

The B data edit descriptor transfers binary (base 2) values. It takes the following form:

```
Bw[.m]
```

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the B data edit descriptor transfers *w* characters from an external field and assigns their binary value to the corresponding I/O list item. The external field must contain only binary digits (0 or 1) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the B edit descriptor:

Format	Input	Value
B4	1001	9
B1	1	1
B2	0	0

Rules for Output Processing

On output, the B data edit descriptor transfers the binary value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of binary digits) with no leading zeros. A negative value is transferred in internal form.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the B edit descriptor:

Format	Value	Output
B4	9	1001
B2	0	$0

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.3.3. O Editing

The O data edit descriptor transfers octal (base 8) values. It takes the following form:

```
Ow[.m]
```

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the O data edit descriptor transfers *w* characters from an external field and assigns their octal value to the corresponding I/O list item. The external field must contain only octal digits (0 through 7) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the O edit descriptor:

Format	Input	Value
O5	32767	32767
O4	16234	1623
O3	97Δ	An error; the 9 is invalid in octal notation

Rules for Output Processing

On output, the O data edit descriptor transfers the octal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of octal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the O edit descriptor:

Format	Value	Output
O6	32767	Δ77777
O12	–32767	Δ37777700001
O2	14261	**
O4	27	ΔΔ33
O5	10.5	41050
O4.2	7	ΔΔ07
O4.4	7	0007

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.3.4. Z Editing

The Z data edit descriptor transfers hexadecimal (base 16) values. It takes the following form:

```
Zw[.m]
```

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the Z data edit descriptor transfers *w* characters from an external field and assigns their hexadecimal value to the corresponding I/O list item. The external field must contain only hexadecimal digits (0 though 9 and A (a) through F(f)) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the Z edit descriptor:

Format	Input	Value
Z3	A94	A94
Z5	A23DEF	A23DE
Z5	95.AF2	An error; the decimal point is invalid

Rules for Output Processing

On output, the Z data edit descriptor transfers the hexadecimal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of hexadecimal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the Z edit descriptor:

Format	Value	Output
Z4	32767	7FFF
Z9	-32767	ΔFFFF8001
Z2	16	10
Z4	-10.5	****
Z3.3	2708	A94
Z6.4	2708	ΔΔ0A94

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.4. Real and Complex Editing

Real and complex editing is controlled by the F, E, D, EN, ES, and G data edit descriptors.

If no field width (*w*) is specified for a real data edit descriptor, the system supplies default values.

Real data edit descriptors can be affected by specified scale factors.

Note

Do not use the real data edit descriptors when attempting to parse textual input. These descriptors accept some forms that are purely textual as valid numeric input values. For example, input values T and F are treated as values -1.0 and 0.0, respectively, for .TRUE. and .FALSE..

For More Information:

- On the scale factor, see Section 11.4.5.
- On system default values for data edit descriptors , see Section 11.3.7.
- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.4.1. F Editing

The F data edit descriptor transfers real values. It takes the following form:

`Fw.d`

The value of *d* (the number of places after the decimal point) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the F data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The external field data must be an integer or real constant.

If the input field contains only an exponent letter or decimal point, it is treated as a zero value.

If the input field does not contain a decimal point or an exponent, it is treated as a real number of *w* digits, with *d* digits to the right of the decimal point. (Leading zeros are added, if necessary.)

If the input field contains a decimal point, the location of that decimal point overrides the location specified by the F descriptor.

If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

The following shows input using the F edit descriptor:

Format	**Input**	**Value**
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

Rules for Output Processing

On output, the F data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The *w* must be greater than or equal to *d*+3 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- At least one digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point

The following shows output using the F edit descriptor:

Format	**Value**	**Output**
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.1	51.44	**

Format	Value	Output
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	******
F5.2	-.2	-0.20

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.4.2. E and D Editing

The E and D data edit descriptors transfer real values in exponential form. They take the following form:

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

For the E edit descriptor, the value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

For the D edit descriptor, the value of *d* must not exceed the value of *w*.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the E and D data edit descriptors transfer *w* characters from an external field and assigns their real value to the corresponding I/O list item. The E and D descriptors interpret and assign input data in the same way as the F data edit descriptor (see Section 11.3.4.1).

The following shows input using the E and D edit descriptors:

Format	Input	Value
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E-6	1022.43E-6
E15.3	52.3759663ΔΔΔΔΔ	52.3759663
E12.5	210.5271D+10[1]	210.5271E10
BZ,D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

[1] If the I/O list item is single-precision real, the E edit descriptor treats the D exponent indicator as an E indicator.

Rules for Output Processing

On output, the E and D data edit descriptors transfer the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)

- An optional zero to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
Ew.d	|exp | ≤ 99	E+nn	E–nn
	99 < |exp | ≤ 999	+nnn	–nnn
Ew.dEe	|exp | ≤ $10^{e} - 1$	E+$n_1n_2...n_e$	E–$n_1n_2...n_e$
Dw.d	|exp | ≤ 99	D+nn or E+nn	D–nn or E–nn
	99 < |exp | ≤ 999	+nnn	–nnn

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional for the E edit descriptor; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+ *e*+5.

Note

The *w* can be as small as *d*+5 or *d*+ *e*+3, if the optional fields for the sign and the zero are omitted.

The following shows output using the E and D edit descriptors:

Format	Value	Output
E11.2	475867.222	ΔΔΔ0.48E+06
E11.5	475867.222	0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E13.3E6	0.000123	0.123E-000003
D14.3	0.0363	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	*********

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.
- On the scale factor, see Section 11.4.5.

11.3.4.3. EN Editing

The EN data edit descriptor transfers values by using engineering notation. It takes the following form:

```
ENw.d[Ee]
```

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the EN data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The EN descriptor interprets and assigns input data in the same way as the F data edit descriptor (see Section 11.3.4.1).

The following shows input using the EN edit descriptor:

Format	Input	Value
EN11.3	ΔΔ5.321E+00	5.32100
EN11.3	–600.00E-03	-.60000
EN12.3	ΔΔΔ3.150E-03	.00315
EN12.3	ΔΔΔ3.829E+03	3829.0

Rules for Output Processing

On output, the EN data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in engineering notation, where the decimal exponent is divisible by 3 and the absolute value of the significant is greater than or equal to 1 and less than 1000 (unless the output value is zero).

The *w* should be greater than or equal to *d*+9 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One to three digits to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ENw.d	|exp | ≤ 99	E+nn	E–nn
	99 < |exp | ≤ 999	+nnn	–nnn
ENw.dEe	|exp | ≤ $10^e - 1$	E+$n_1n_2...n_e$	E–$n_1n_2...n_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+ *e*+5.

The following shows output using the EN edit descriptor:

Format	Value	Output
EN11.2	475867.222	Δ475.87E+03
EN11.5	475867.222	**********
EN12.3	0.00069	Δ690.000E-06
EN10.3	-0.5555	**********
EN11.2	0.0	Δ000.00E-03

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.4.4. ES Editing

The ES data edit descriptor transfers values by using scientific notation. It takes the following form:

```
ESw.d[Ee]
```

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the ES data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The ES descriptor interprets and assigns input data in the same way as the F data edit descriptor (see Section 11.3.4.1).

The following shows input using the ES edit descriptor:

Format	Input	Value
ES11.3	ΔΔ5.321E+00	5.32100
ES11.3	-6.000E-01	-.60000
ES12.3	ΔΔΔ3.150E-03	.00315
ES12.3	ΔΔΔ3.829E+03	3829.0

Rules for Output Processing

On output, the ES data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in scientific notation, where the absolute value of the significand is greater than or equal to 1 and less than 10 (unless the output value is zero).

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ESw.d	|exp| ≤ 99	E+nn	E–nn
	99 < |exp| ≤ 999	+nnn	–nnn
ESw.dEe	|exp| ≤ $10^{e} - 1$	E+n $_1$n $_2$...n $_e$	E–n $_1$n $_2$...n $_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+ *e*+5.

The following shows output using the ES edit descriptor:

Format	Value	Output
ES11.2	473214.356	ΔΔΔ4.73E+05
ES11.5	473214.356	4.73214E+05
ES12.3	0.00069	ΔΔΔ6.900E-04
ES10.3	-.5555	-5.555E-01
ES11.2	0.0	Δ0.000E+00

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.

11.3.4.5. G Editing

The G data edit descriptor generally transfers values of real type, but it can be used to transfer values of any intrinsic type. It takes the following form:

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

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item can be of any intrinsic type.

When used to specify I/O for integer, logical, or character data, the edit descriptor follows the same rules as I *w*, L *w*, and A *w*, respectively, and *d* and *e* have no effect.

Rules for Real Input Processing

On input, the G data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The G descriptor interprets and assigns input data in the same way as the F data edit descriptor (see Section 11.3.4.1).

Rules for Real Output Processing

On output, the G data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The form in which the value is written is a function of the magnitude of the value, as described in Table 11.2.

Table 11.2. Effect of Data Magnitude on G Format Conversions

Data Magnitude	Equivalent Conversion
$0 < m < 0.1 - 0.5 \times 10^{-d-1}$	Ew.d[Ee]
m = 0	F(w −n).(d −1), n('b')
$0.1 - 0.5 \times 10^{-d-1} \leq m < 1 - 0.5 \times 10^{-d}$	F(w −n).d, n('b')
$1 - 0.5 \times 10^{-d} \leq m < 10 - 0.5 \times 10^{-d+1}$	F(w −n).(d −1), n('b')
$10 - 0.5 \times 10^{-d+1} \leq m < 100 - 0.5 \times 10^{-d+2}$	F(w −n).(d −2), n('b')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2} \leq m < 10^{d-1} - 0.5 \times 10^{-1}$	F(w −n).1, n('b')
$10^{d-1} - 0.5 \times 10^{-1} \leq m < 10^{d} - 0.5$	F(w −n).0, n('b')
$m \geq 10^{d} - 0.5$	Ew.d[Ee]

The 'b' is a blank following the numeric data representation. For Gw.d, n('b') is 4 blanks. For Gw.dEe, n('b') is *e*+2 blanks.

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The 4-digit or *e*+2-digit exponent

If *e* is specified, the *w* should be greater than or equal to *d*+ *e*+5.

The following shows output using the G edit descriptor and compares it to output using equivalent F editing:

Value	Format	Output with G	Format	Output with F
0.01234567	G13.6	Δ0.123457E-01	F13.6	ΔΔΔΔΔ0.012346

Value	Format	Output with G	Format	Output with F
-0.12345678	G13.6	-0.123457ΔΔΔΔ	F13.6	ΔΔΔΔ-0.123457
1.23456789	G13.6	ΔΔ1.23457ΔΔΔΔ	F13.6	ΔΔΔΔΔ1.234568
12.34567890	G13.6	ΔΔ12.3457ΔΔΔΔ	F13.6	ΔΔΔΔ12.345679
123.45678901	G13.6	ΔΔ123.457ΔΔΔΔ	F13.6	ΔΔΔ123.456789
-1234.56789012	G13.6	Δ-1234.57ΔΔΔΔ	F13.6	Δ-1234.567890
12345.67890123	G13.6	ΔΔ12345.7ΔΔΔΔ	F13.6	Δ12345.678901
123456.78901234	G13.6	ΔΔ123457.ΔΔΔΔ	F13.6	123456.789012
-1234567.89012345	G13.6	-0.123457E+07	F13.6	**********

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.
- On the I data edit descriptor, see Section 11.3.3.1.
- On the L data edit descriptor, see Section 11.3.5.
- On the A data edit descriptor, see Section 11.3.6.
- On the scale factor, see Section 11.4.5.

11.3.4.6. Complex Editing

A complex value is an ordered pair of real values. Complex editing is specified by a pair of real edit descriptors, using any combination of the forms: Fw.d, Ew.d[Ee], Dw.d, ENw.d[Ee], ESw.d[Ee], or Gw.d[Ee].

Rules for Input Processing

On input, the two successive fields are read and assigned to the corresponding complex I/O list item as its real and imaginary part, respectively.

The following shows input using complex editing:

Format	Input	Value
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 123456.789

Rules for Output Processing

On output, the two parts of the complex value are transferred under the control of repeated or successive real edit descriptors. The two parts are transferred consecutively without punctuation or blanks, unless control or character string edit descriptors are specified between the pair of real edit descriptors.

The following shows output using complex editing:

Format	Value	Output
2F8.5	2.3547188, 3.456732	$2.35472 $3.45673

Format	Value	Output
E9.2, 'Δ,Δ',E5.3	47587.222, 56.123	$0.48E+06$,Δ*****

For More Information:

- On the form for data edit descriptors, see Section 11.3.1.
- On general rules for numeric editing, see Section 11.3.2.
- On complex constants, see Section 3.2.3.1.

11.3.5. Logical Editing (L)

The L data edit descriptor transfers logical values. It takes the following form:

```
Lw
```

The specified I/O list item must be of type logical or integer.

The G edit descriptor can be used to edit logical data; it follows the same rules as L *w*.

Rules for Input Processing

On input, the L data edit descriptor transfers *w* characters from an external field and assigns their logical value to the corresponding I/O list item. The value assigned depends on the external field data, as follows:

- .TRUE. is assigned if the first nonblank character is .T, T, .t, or t. The logical constant .TRUE. is an acceptable input form.
- .FALSE. is assigned if the first nonblank character is .F, F. .f, or f, or the entire field is filled with blanks. The logical constant .FALSE. is an acceptable input form.

If an other value appears in the external field, an error occurs.

Rules for Output Processing

On output, the L data edit descriptor transfers the following to an external field that is *w* characters long: w − 1 blanks, followed by a T or F (if the value is .TRUE. or .FALSE., respectively).

The following shows output using the L edit descriptor:

Format	Value	Output
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

For More Information:

On the general form for data edit descriptors, see Section 11.3.1.

11.3.6. Character Editing (A)

The A data edit descriptor transfers character or Hollerith values. It takes the following form:

`A[w]`

If the corresponding I/O list item is of type character, character data is transferred. If the list item is of any other type, Hollerith data is transferred.

The G edit descriptor can be used to edit character data; it follows the same rules as A *w*.

Rules for Input Processing

On input, the A data edit descriptor transfers *w* characters from an external field and assigns them to the corresponding I/O list item.

The maximum number of characters that can be stored depends on the size of the I/O list item, as follows:

- For character data, the maximum size is the length of the corresponding I/O list item.
- For noncharacter data, the maximum size depends on the data type, as shown in Table 11.3.

Table 11.3. Size Limits for Noncharacter Data Using A Editing

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL(1) or LOGICAL*1	1
LOGICAL(2) or LOGICAL*2	2
LOGICAL(4) or LOGICAL*4	4
LOGICAL(8) or LOGICAL*8	8
INTEGER(1) or INTEGER*1	1
INTEGER(2) or INTEGER*2	2
INTEGER(4) or INTEGER*4	4
INTEGER(8) or INTEGER*8	8
REAL(4) or REAL*4	4
DOUBLE PRECISION	8
REAL(8) or REAL*8	8
REAL(16) or REAL*16	16
COMPLEX(4) or COMPLEX*8	8[1]
DOUBLE COMPLEX	16[1]
COMPLEX(8) or COMPLEX*16	16[1]
COMPLEX(16) or COMPLEX*32	32[1]

[1]Complex values are treated as pairs of real numbers, so complex editing requires a pair of real edit descriptors. (See Section 11.3.4.6).

If *w* is equal to or greater than the length (*len*) of the input item, the rightmost characters are assigned to that item. The leftmost excess characters are ignored.

If *w* is less than *len*, or less than the number of characters that can be stored, *w* characters are assigned to the list item, left-justified, and followed by trailing blanks.

The following shows input using the A edit descriptor:

Format	Input	Value	Data Type
A6	PAGEΔ#	#	CHARACTER(LEN=1)
A6	PAGEΔ#	EΔ#	CHARACTER(LEN=3)
A6	PAGEΔ#	PAGEΔ#	CHARACTER(LEN=6)
A6	PAGEΔ#	PAGEΔ#ΔΔ	CHARACTER(LEN=8)
A6	PAGEΔ#	#	LOGICAL(1)
A6	PAGEΔ#	Δ#	INTEGER(2)
A6	PAGEΔ#	GEΔ#	REAL(4)
A6	PAGEΔ#	PAGEΔ#ΔΔ	REAL(8)

Rules for Output Processing

On output, the A data edit descriptor transfers the contents of the corresponding I/O list item to an external field that is *w* characters long.

If *w* is greater than the size of the list item, the data is transferred to the output field, right-justified, with leading blanks. If *w* is less than or equal to the size of the list item, the leftmost *w* characters are transferred.

The following shows output using the A edit descriptor:

Format	Value	Output
A5	OHMS	ΔOHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

11.3.7. Default Widths for Data Edit Descriptors

If *w* (the field width) is omitted for the data edit descriptors, the system applies default values. For the real data edit descriptors, the system also applies default values for *d* (the number of characters to the right of the decimal point), and *e* (the number of characters in the exponent).

These defaults are based on the data type of the I/O list item, and are listed in Table 11.4.

Table 11.4. Default Widths for Data Edit Descriptors

Edit Descriptor	Data Type of I/O List Item	w
I, B, O, Z, G	BYTE	7
	INTEGER(1), LOGICAL(1)	7
	INTEGER(2), LOGICAL(2)	7
	INTEGER(4), LOGICAL(4)	12
	INTEGER(8), LOGICAL(8)	23
O, Z	REAL(4)	12
	REAL(8)	23
	REAL(16)	44
	CHARACTER*len	MAX(7, 3*len)

Edit Descriptor	Data Type of I/O List Item	w
L, G	LOGICAL(1), LOGICAL(2) LOGICAL(4), LOGICAL(8)	2
F, E, EN, ES, G, D	REAL(4), COMPLEX(4)	15 **d**:7 **e**:2
	REAL(8), COMPLEX(8)	25 **d**:16 **e**:2
	REAL(16), COMPLEX(16)	42 **d**:33 **e**:3
A[1], G	LOGICAL(1)	1
	LOGICAL(2), INTEGER(2)	2
	LOGICAL(4), INTEGER(4)	4
	LOGICAL(8), INTEGER(8)	8
	REAL(4), COMPLEX(4)	4
	REAL(8), COMPLEX(8)	8
	REAL(16), COMPLEX(16)	16
	CHARACTER*len	len

[1]The default is the actual length of the corresponding I/O list item.

11.3.8. Terminating Short Fields of Input Data

On input, an edit descriptor such as Fw.d specifies that *w* characters (the field width) are to be read from the external field.

If the field contains fewer than *w* characters, the input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data.

Padding Short Fields

You can use the OPEN statement specifier PAD= 'YES' to indicate blank padding for short fields of input data. However, blanks can be interpreted as blanks *or* zeros, depending on which default behavior is in effect at the time. Consider the following:

```
READ (2, '(I5)') J
```

If 3 is input for J, the value of J will be 30000 or 3 depending on which default behavior is in effect (BLANK='NULL' or BLANK='ZERO'). This can give unexpected results.

To ensure that the desired behavior is in effect, explicitly specify the BN or BZ edit descriptor. For example, the following ensures that blanks are interpreted as blanks (and not as zeros):

```
READ (2, '(BN, I5)') J
```

Using Commas to Separate Input Data

You can use a comma to terminate a short data field. The comma has no effect on the d part (the number of characters to the right of the decimal point) of the specification.

The comma overrides the w specified for the I, B, O, Z, F, E, D, EN, ES, G, and L edit descriptors. For example, suppose the following statements are executed:

```
     READ (5,100) I,J,A,B
```

```
100  FORMAT (2I6,2F10.2)
```

Suppose a record containing the following values is read:

```
1, -2, 1.0, 35
```

The following assignments occur:

```
I = 1
J = -2
A = 1.0
B = 0.35
```

A comma can only terminate fields less than w characters long. If a comma follows a field of w or more characters, the comma is considered part of the next field.

A null (zero-length) field is designated by two successive commas, or by a comma after a field of w characters. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.0D0, 0.0Q0, or .FALSE.

For More Information:

On input processing, see Section 11.3.2.

11.4. Control Edit Descriptors

A control edit descriptor either directly determines how text is displayed or affects the conversions performed by subsequent data edit descriptors.

This section describes the forms for control edit descriptors and the individual descriptors themselves.

11.4.1. Forms for Control Edit Descriptors

A control edit descriptor takes one of the following forms:

```
c
cn
nc
```

c

Is one of the following format codes: T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, \, Δ, and Q.

n

Is a number of character positions. It must be a positive integer literal constant; or variable format expression; no kind parameter can be specified. It cannot be a named constant.

The range of *n* is 1 through 2147483647 (2**31–1). Actual useful ranges may be constrained by record sizes (RECL) and the file system.

Rules and Behavior

In general, control edit descriptors are nonrepeatable. The only exception is the slash (/) edit descriptor, which can be preceded by a repeat specification.

The control edit descriptors have the following specific forms:

Positional:	Tn, TLn, TRn, and nX
Sign:	S, SP, and SS
Blank interpretation:	BN and BZ
Scale factor:	kP
Miscellaneous:	:, /, \, Δ, and Q

The P edit descriptor is an exception to the general control edit descriptor syntax. It is preceded by a scale factor, rather than a character position specifier.

Control edit descriptors can be grouped in parentheses and preceded by a group repeat specification.

For More Information:

- On format specifications, in general, see Section 11.2.
- On group repeat specifications, see Section 11.6.

11.4.2. Positional Editing

The T, TL, TR, and X edit descriptors specify the position where the next character is transferred to or from a record.

On output, these descriptors do not themselves cause characters to be transferred and do not affect the length of the record. If characters are transferred to positions at or after the position specified by one of these descriptors, positions skipped and not previously filled are filled with blanks. The result is as if the entire record was initially filled with blanks.

The TR and X edit descriptors produce the same results.

11.4.2.1. T Editing

The T edit descriptor specifies a character position in an I/O record. It takes the following form:

```
Tn
```

The *n* is a positive integer literal constant (with no kind parameter) indicating the character position of the record, relative to the left tab limit.

On input, the T descriptor positions the external record at the character position specified by *n*. On output, the T descriptor indicates that data transfer begins at the *n*th character position of the external record.

Examples

Suppose a file has a record containing the value `ABC$ΔΔΔ$XYZ`, and the following statements are executed:

```
     READ (11,10) VALUE1, VALUE2
10   FORMAT (T7,A3,T1,A3)
```

The values read first are XYZ, then ABC.

Suppose the following statements are executed:

```
     PRINT 25
25   FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

The following line is printed at the positions indicated:

```
Position 20                    Position 50
  ↓                           ↓
  COLUMN 1                    COLUMN 2
```

Note that the first character of the record printed was reserved as a control character. (For more information, see Section 11.8).

11.4.2.2. TL Editing

The TL edit descriptor specifies a character position to the *left* of the current position in an I/O record. It takes the following form:

```
TLn
```

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the left of the current character.

If *n* is greater than or equal to the current position, the next character accessed is the first character of the record.

11.4.2.3. TR Editing

The TR edit descriptor specifies a character position to the *right* of the current position in an I/O record. It takes the following form:

```
TRn
```

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

11.4.2.4. X Editing

The X edit descriptor specifies a character position to the right of the current position in an I/O record. It takes the following form:

```
nX
```

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

On output, the X edit descriptor does not output any characters when it appears at the end of a format specification; for example:

```
     WRITE (6,99) K
99   FORMAT ('ΔK=',I6,5X)
```

This example writes a record of only 9 characters. To cause *n* trailing blanks to be output at the end of a record, specify a format of `n('Δ')`.

11.4.3. Sign Editing

The S, SP, and SS edit descriptors control the output of the optional plus (+) sign within numeric output fields. These descriptors have no effect during execution of input statements.

Within a format specification, a sign editing descriptor affects all subsequent I, F, E, EN, ES, D, and G descriptors until another sign editing descriptor occurs.

11.4.3.1. SP Editing

The SP edit descriptor causes the processor to produce a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

```
SP
```

11.4.3.2. SS Editing

The SS edit descriptor causes the processor to suppress a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

```
SS
```

11.4.3.3. S Editing

The S edit descriptor restores the plus sign as optional for all subsequent positive numeric fields. It takes the following form:

```
S
```

The S edit descriptor restores to the processor the discretion of producing plus characters on an optional basis.

11.4.4. Blank Editing

The BN and BZ descriptors control the interpretation of embedded and trailing blanks within numeric input fields. These descriptors have no effect during execution of output statements.

Within a format specification, a blank editing descriptor affects all subsequent I, B, O, Z, F, E, EN, ES, D, and G descriptors until another blank editing descriptor occurs.

The blank editing descriptors override the effect of the BLANK specifier during execution of a particular input data transfer statement. (For more information on the BLANK specifier in OPEN statements, see Section 12.6.4).

11.4.4.1. BN Editing

The BN edit descriptor causes the processor to ignore all embedded and trailing blanks in numeric input fields. It takes the following form:

```
BN
```

The input field is treated as if all blanks have been removed and the remainder of the field is right-justified. An all-blank field is treated as zero.

11.4.4.2. BZ Editing

The BZ edit descriptor causes the processor to interpret all embedded and trailing blanks in numeric input fields as zeros. It takes the following form:

```
BZ
```

11.4.5. Scale Factor Editing (P)

The P edit descriptor specifies a scale factor, which moves the location of the decimal point in real values and the two real parts of complex values. It takes the following form:

```
kP
```

The *k* is a signed (sign is optional if positive), integer literal constant specifying the number of positions, to the left or right, that the decimal point is to move (the scale factor). The range of *k* is –128 to 127.

At the beginning of a formatted I/O statement, the value of the scale factor is zero. If a scale editing descriptor is specified, the scale factor is set to the new value, which affects all subsequent real edit descriptors until another scale editing descriptor occurs.

To reinstate a scale factor of zero, you must explicitly specify 0P.

Format reversion does not affect the scale factor. (For more information on format reversion, see Section 11.9).

Rules for Input Processing

On input, a positive scale factor moves the decimal point to the left, and a negative scale factor moves the decimal point to the right. (On output, the effect is the reverse.)

On input, when an input field using an F, E, D, EN, ES, or G real edit descriptor contains an explicit exponent, the scale factor has no effect. Otherwise, the internal value of the corresponding I/O list item is equal to the external field data multiplied by 10^{-k}. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A –2P scale factor multiplies an input value by 100, moving the decimal point two places to the right.

The following shows input using the P edit descriptor:

Format	Input	Value
3PE10.5	ΔΔΔ37.614Δ	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	37614.0

The scale factor must precede the first real edit descriptor associated with it, but it need not immediately precede the descriptor. For example, the following all have the same effect:

```
(3P, I6, F6.3, E8.1)
(I6, 3P, F6.3, E8.1)
(I6, 3PF6.3, E8.1)
```

Note that if the scale factor immediately precedes the associated real edit descriptor, the comma separator is optional.

Rules for Output Processing

On output, a positive scale factor moves the decimal point to the right, and a negative scale factor moves the decimal point to the left. (On input, the effect is the reverse).

On output, the effect of the scale factor depends on which kind of real editing is associated with it, as follows:

- For F editing, the external value equals the internal value of the I/O list item multiplied by 10^k. This changes the magnitude of the data.
- For E and D editing, the external decimal field of the I/O list item is multiplied by 10^k, and *k* is subtracted from the exponent. This changes the form of the data.

  A positive scale factor decreases the exponent; a negative scale factor increases the exponent.

  For a positive scale factor, *k* must be less than *d* + 2 or an output conversion error occurs.
- For G editing, the scale factor has no effect if the magnitude of the data to be output is within the effective range of the descriptor (the G descriptor supplies its own scaling).

  If the magnitude of the data field is outside G descriptor range, E editing is used, and the scale factor has the same effect as E output editing.
- For EN and ES editing, the scale factor has no effect.

The following shows output using the P edit descriptor:

Format	Value	Output
1PE12.3	-270.139	ΔΔ-2.701E+02
1P,E12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-0.03E+04

The following shows a FORMAT statement containing a scale factor:

```
     DIMENSION A(6)
     DO 10 I=1,6
10   A(I) = 25.
     WRITE (6, 100) A
100  FORMAT(' ', F8.2, 2PF8.2, F8.2)
```

The preceding statements produce the following results:

```
    25.00 2500.00 2500.00
  2500.00 2500.00 2500.00
```

11.4.6. Slash Editing (/)

The slash edit descriptor terminates data transfer for the current record and starts data transfer for a new record. It takes the following form:

```
[r]/
```

The *r* is a repeat specification. It must be a positive default integer literal constant; no kind parameter can be specified.

The range of *r* is 1 through 2147483647 (2**31–1). If *r* is omitted, it is assumed to be 1.

Multiple slashes cause the system to skip input records or to output blank records, as follows:

- When *n* consecutive slashes appear between two edit descriptors, *n* − 1 records are skipped on input, or *n* − 1 blank records are output. The first slash terminates the current record. The second slash terminates the first skipped or blank record, and so on.
- When *n* consecutive slashes appear at the beginning or end of a format specification, *n* records are skipped or *n* blank records are output, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, suppose the following statements are specified:

```
      WRITE (6,99)
99    FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

The following lines are written:

```
                    Column 50, top of page
                    ↓
                    HEADING LINE
(blank line)
                    SUBHEADING LINE
(blank line)
(blank line)
```

Note that the first character of the record printed was reserved as a control character (see Section 11.8).

11.4.7. Colon Editing (:)

The colon edit descriptor terminates format control if no more items are in the I/O list. For example, suppose the following statements are specified:

```
      PRINT 1,3
      PRINT 2,13
1     FORMAT (' I=',I2,' J=',I2)
2     FORMAT (' K=',I2,:,' L=',I2)
```

The following lines are written:

```
I=Δ3ΔJ=
K=13
```

If I/O list items remain, the colon edit descriptor has no effect.

11.4.8. Dollar Sign ($) and Backslash (\) Editing

The dollar sign and backslash edit descriptors modify the output of carriage control specified by the first character of the record. They only affect carriage control for formatted files, and have no effect on input.

If the first character of the record is a blank or a plus sign (+), the dollar sign and backslash descriptors suppress carriage return (after printing the record).

For terminal device I/O, when this trailing carriage return is suppressed, a response follows output on the same line. For example, suppose the following statements are specified:

```
     TYPE 100
100  FORMAT (' ENTER RADIUS VALUE ',$)
     ACCEPT 200, RADIUS
200  FORMAT (F6.2)
```

The following prompt is displayed:

```
ENTER RADIUS VALUE
```

Any response (for example, “12.”) is then displayed on the same line:

```
ENTER RADIUS VALUE    12.
```

If the first character of the record is 0, 1, or ASCII NUL, the dollar sign and backslash descriptors have no effect.

Consider the following:

```
     CHARACTER(20) MYNAME
     WRITE (*,9000)
9000 FORMAT ('0Please type your name:',\)
     READ  (*,9001) MYNAME
9001 FORMAT (A20)
     WRITE (*,9002) ' ',MYNAME
9002 FORMAT (1X,A20)
```

This example advances two lines, prompts for input, awaits input on the same line as the prompt, and prints the input.

11.4.9. Character Count Editing (Q)

The character count edit descriptor returns the remaining number of characters in the current input record.

The corresponding I/O list item must be of type integer or logical. For example, suppose the following statements are specified:

```
     READ (4,1000) XRAY, KK, NCHRS, (ICHR(I), I=1,NCHRS)
1000 FORMAT (E15.7,I4,Q,(80A1))
```

Two fields are read into variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS, and exactly that many characters are read into the array ICHR. (This instruction can fail if the record is longer than 80 characters).

If you place the character count descriptor first in a format specification, you can determine the length of an input record.

On output, the character count edit descriptor causes the corresponding I/O list item to be skipped.

11.5. Character String Edit Descriptors

Character string edit descriptors control the output of character strings. The character string edit descriptors are the character constant and H edit descriptor.

Although no string edit descriptor can be preceded by a repeat specification, a parenthesized group of string edit descriptors can be preceded by a repeat specification (see Section 11.6).

11.5.1. Character Constant Editing

The character constant edit descriptor causes a character string to be output to an external record. It takes one of the following forms:

```
'string'
"string"
```

The *string* is a character literal constant; no kind parameter can be specified. Its length is the number of characters between the delimiters; two consecutive delimiters are counted as one character.

To include an apostrophe in a character constant that is enclosed by apostrophes, place two consecutive apostrophes (' ') in the format specification; for example:

```
 50   FORMAT ('TODAY''SΔDATEΔIS:Δ',I2,'/',I2,'/',I2)
```

Similarly, to include a quotation mark in a character constant that is enclosed by quotation marks, place two consecutive quotation marks (" ") in the format specification.

For More Information:

- On format specifications, in general, see Section 11.2.
- On character constants, see Character Constants in Section 3.2.5.

11.5.2. H Editing

The H edit descriptor transfers data between the external record and the H edit descriptor itself. The H edit descriptor is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. VSI Fortran fully supports features deleted in Fortran 95.

An H edit descriptor has the form of a Hollerith constant, as follows:

```
nHstring
```

n

Is an unsigned, positive default integer literal constant (with no kind parameter) indicating the number of characters in *string* (including blanks and tabs).

The range of *n* is 1 through 2147483647 (2**31–1). Actual useful ranges may be constrained by record sizes (RECL) and the file system.

string

Is a string of printable ASCII characters.

On input, the H edit descriptor transfers *n* characters from the external field to the edit descriptor. The first character appears immediately after the letter H. Any characters in the edit descriptor before input are replaced by the input characters.

On output, the H edit descriptor causes *n* characters following the letter H to be output to an external record.

For More Information:

- On format specifications, in general, see Section 11.2.

- On obsolescent features in Fortran 95 and Fortran 90, see Appendix A.

11.6. Nested and Group Repeat Specifications

Format specifications can include nested format specifications enclosed in parentheses; for example:

```
15   FORMAT (E7.2,I8,I2,(A5,I6))

35   FORMAT (A6,(L8(3I2)),A)
```

A group repeat specification can precede a nested group of edit descriptors. For example, the following statements are equivalent, and the second statement shows a group repeat specification:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,I5,I5)

50   FORMAT (2I8,3(F8.3,E15.7),2I5)
```

If a nested group does not show a repeat count, a default count of 1 is assumed.

Normally, the string edit descriptors and control edit descriptors cannot be repeated (except for slash), but any of these descriptors can be enclosed in parentheses and preceded by a group repeat specification. For example, the following statements are valid:

```
76   FORMAT ('MONTHLY',3('TOTAL'))

100  FORMAT (I8,4(T7),A4)
```

For More Information:

- On repeat specifications for data edit descriptors, see Section 11.3.1.
- On group repeat specifications and format reversion, see Section 11.9.

11.7. Variable Format Expressions

A variable format expression is a numeric expression enclosed in angle brackets (<>) that can be used in a FORMAT statement or in a character format specification.

The numeric expression can be any valid Fortran expression, including function calls and references to dummy arguments.

If the expression is not of type integer, it is converted to integer type before being used.

If the value of a variable format expression does not obey the restrictions on magnitude applying to its use in the format, an error occurs.

Variable format expressions cannot be used with the H edit descriptor, and they are not allowed in character format specifications.

Variable format expressions are evaluated each time they are encountered in the scan of the format. If the value of the variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Examples

Consider the following statement:

```
FORMAT (I<J+1>)
```

When the format is scanned, the preceding statement performs an I (integer) data transfer with a field width of J+1. The expression is reevaluated each time it is encountered in the normal format scan.

Consider the following statements:

```
     DIMENSION A(5)
     DATA A/1.,2.,3.,4.,5./

     DO 10 I=1,10
     WRITE (6,100) I
100  FORMAT (I<MAX(I,5)>)
10   CONTINUE

     DO 20 I=1,5
     WRITE (6,101) (A(I), J=1,I)
101  FORMAT (<I>F10.<I-1>)
20   CONTINUE
     END
```

On execution, these statements produce the following output:

```
   1
   2
   3
   4
   5
    6
     7
      8
       9
       10
       1.
      2.0        2.0
     3.00       3.00     3.00
    4.000      4.000    4.000     4.000
   5.0000     5.0000    5.0000    5.0000    5.0000
```

For More Information:

On the synchronization of I/O lists with formats, see Section 11.9.

11.8. Printing of Formatted Records

On output, if a file was opened with CARRIAGECONTROL= 'FORTRAN' in effect or the file is being processed by the `fortpr` format utility, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but used to control vertical spacing.

Table 11.5 lists the valid control characters for printing.

Table 11.5. Control Characters for Printing

Character	Meaning	Effect
+	Overprinting	Outputs the record (at the current position in the current line) and a carriage return.

Character	Meaning	Effect
Δ	One line feed	Outputs the record (at the beginning of the following line) and a carriage return.
0	Two line feeds	Outputs the record (after skipping a line) and a carriage return.
1	Next page	Outputs the record (at the beginning of a new page) and a carriage return.
$	Prompting	Outputs the record (at the beginning of the following line), but no carriage return.
ASCII NUL[1]	Overprinting with no advance	Outputs the record (at the current position in the current line), but no carriage return.

[1]Specify as CHAR (0).

Any other character is interpreted as a blank and is deleted from the print line. If you do not specify a control character for printing, the first character of the record is not printed.

11.9. Interaction Between Format Specifications and I/O Lists

Format control begins with the execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next item in the I/O list (if one exists) and the next edit descriptor in the format specification.

Both the I/O list and the format specification are interpreted from left to right, unless repeat specifications or implied-do lists appear.

If an I/O list specifies at least one list item, at least one data edit descriptor (I, B, O, Z, F, E, EN, ES, D, G, L, or A) or the Q edit descriptor must appear in the format specification; otherwise, an error occurs.

Each data edit descriptor (or Q edit descriptor) corresponds to one item in the I/O list, except that an I/O list item of type complex requires the interpretation of two F, E, EN, ES, D, or G edit descriptors. No I/O list item corresponds to a control edit descriptor (X, P, T, TL, TR, SP, SS, S, BN, BZ, Δ, or :), or a character string edit descriptor (H and character constants). For character string edit descriptors, data transfer occurs directly between the external record and the format specification.

When format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding I/O list item specified. If there is such an item, it is transferred under control of the edit descriptor, and then format control proceeds. If there is no corresponding I/O list item, format control terminates.

If there are no other I/O list items to be processed, format control also terminates when the following occurs:

- A colon edit descriptor is encountered.
- The end of the format specification is reached.

If additional I/O list items remain, part or all of the format specification is reused in format reversion.

In format reversion, the current record is terminated and a new one is initiated. Format control then reverts to one of the following (in order) and continues from that point:

1. The group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification

2. The initial opening parenthesis of the format specification

Format reversion has no effect on the scale factor, the sign control edit descriptors (S, SP, or SS), or the blank interpretation edit descriptors (BN or BZ).

Examples

The data in file FOR002.DAT is to be processed 2 records at a time. Each record starts with a number to be put into an element of a vector B, followed by 5 numbers to be put in a row in matrix A.

FOR002.DAT contains the following data:

```
001 0101 0102 0103 0104 0105
002 0201 0202 0203 0204 0205
003 0301 0302 0303 0304 0305
004 0401 0402 0403 0404 0405
005 0501 0502 0503 0504 0505
006 0601 0602 0603 0604 0605
007 0701 0702 0703 0704 0705
008 0801 0802 0803 0804 0805
009 0901 0902 0903 0904 0905
010 1001 1002 1003 1004 1005
```

Example 11.1 shows how several different format specifications interact with I/O lists to process data in file FOR002.DAT.

Example 11.1. Interaction Between Format Specifications and I/O Lists

```
        INTEGER I, J, A(2,5), B(2)

        OPEN (unit=2, access='sequential', file='FOR002.DAT')

❶        READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2)
❷  100   FORMAT (2 (I3, X, 5(I4,X), /) )

❸        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
   999   FORMAT (' B is ', 2(I3, X), ';  A is', /
        1       (' ', 5 (I4, X)) )

❹        READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2)
   200   FORMAT (2 (I3, X, 5(I4,X), :/) )

❺        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)


❻        READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2)
   300   FORMAT ( (I3, X, 5(I4,X)) )

❼        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)

❽        READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2)
   400   FORMAT ( I3, X, 5(I4,X) )

❾        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
```

```
      END
```

❶ This statement reads B(1); then A(1,1) through A(1,5); then B(2) and A(2,1) through A(2,5).

The first record read (starting with 001) starts the processing of the I/O list.

❷ There are two records, each in the format I3, X, 5(I4, X). The slash (/) forces the reading of the second record after A(1,5) is processed. It also forces the reading of the third record after A(2,5) is processed; no data is taken from that record.

❸ This statement produces the following output:

```
B is   1   2 ;  A is
 101  102  103  104  105
 201  202  203  204  205
```

❹ This statement reads the record starting with 004. The slash (/) forces the reading of the next record after A(1,5) is processed. The colon (:) stops the reading after A(2,5) is processed, but before the slash (/) forces another read.

❺ This statement produces the following output:

```
B is   4   5 ;  A is
 401  402  403  404  405
 501  502  503  504  505
```

❻ This statement reads the record starting with 006. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I3.

❼ This statement produces the following output:

```
B is   6   7 ;  A is
 601  602  603  604  605
 701  702  703  704  705
```

❽ This statement reads the record starting with 008. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I4.

❾ This statement produces the following output:

```
B is   8  90 ;  A is
 801  802  803  804  805
 9010 9020 9030 9040 100
```

The record 009 0901 0902 0903 0904 0905 is processed with I4 as “009 ” for B(2), which is 90. X skips the next “0”. Then “901 ” is processed for A(2,1), which is 9010, “902 ” for A(2,2), “903 ” for A(2,3), and “904 ” for A(2,4). The repeat specification of 5 is now exhausted and the format ends. Format reversion causes another record to be read and starts format processing at the left parenthesis before the I4, so “010 ” is read for A(2,5), which is 100.

For More Information:

- On data edit descriptors, see Section 11.3.
- On control edit descriptors, see Section 11.4.
- On the Q edit descriptor, see Section 11.4.9.
- On character string edit descriptors, see Section 11.5.
- On the scale factor, see Section 11.4.5.

Chapter 12. File Operation I/O Statements

This chapter contains information on the following file connection, inquiry, and positioning statements:

- BACKSPACE (Section 12.1)

  Positions a sequential file at the beginning of the preceding record.

- CLOSE (Section 12.2)

  Terminates the connection between a logical unit and a file or device.

- DELETE (Section 12.3)

  Deletes a record from a relative or indexed file.

- ENDFILE (Section 12.4)

  For sequential files, writes an end-of-file record to the file and positions the file after this record. For direct access files, truncates the file after the current record.

- INQUIRE (Section 12.5)

  Requests information on the status of specified properties of a file or logical unit.

- OPEN (Section 12.6)

  Connects a Fortran logical unit to a file or device; declares attributes for read and write operations.

- REWIND (Section 12.7)

  Positions a sequential file to the beginning of that file.

- UNLOCK (Section 12.8)

  Frees a record in a sequential, relative, or indexed file that was locked by a previous READ statement.

For More Information:

- On data transfer I/O statements, see Chapter 10.
- On control specifiers, see Section 10.2.1.
- On record position, advancement, and transfer, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.1. BACKSPACE Statement

The BACKSPACE statement positions a sequential file at the beginning of the preceding record, making it available for subsequent I/O processing. It takes one of the following forms:

```
BACKSPACE ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var])
```

```
BACKSPACE io-unit
```

io-unit

Is an external unit specifier.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The I/O unit number must specify an open file on disk or magnetic tape.

A BACKSPACE statement must not be specified for a file that is open for direct, append, or keyed access, because record *n* is not available to the RMS I/O system.

If a file is already positioned at the beginning of a file, a BACKSPACE statement has no effect.

Examples

The following statement repositions the file connected to I/O unit 4 back to the preceding record:

```
BACKSPACE 4
```

Consider the following statement:

```
BACKSPACE (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 back to the preceding record. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On append access, see Section 12.6.1.
- On record position, advancement, and transfer, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.2. CLOSE Statement

The CLOSE statement disconnects a file from a unit. It takes the following form:

```
CLOSE ([UNIT=]io-unit [,{STATUS | DISPOSED | DISP} =p] [,ERR=label]
 [,IOSTAT=i-var])
```

io-unit

Is an external unit specifier.

p

Is a scalar default character expression indicating the status of the file after it is closed. It has one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.[1]
'PRINT' [2]	Submits the file to the line printer spooler, then retains it.
'PRINT/DELETE'[2]	Submits the file to the line printer spooler, then deletes it.
'SUBMIT'	Submits the file to the batch job queue, then retains it.
'SUBMIT/ DELETE'	Submits the file to the batch job queue, then deletes it.

[1]Unless OPEN(READONLY) is in effect.

[2]Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is'KEEP'.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The CLOSE statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT specifier is optional if the unit specifier is the first item in the I/O control list.

The status specified in the CLOSE statement supersedes the status specified in the OPEN statement, except that a file opened as a scratch file cannot be saved, printed, or submitted, and a file opened for read-only access cannot be deleted.

If a CLOSE statement is specified for a unit that is not open, it has no effect.

Examples

Consider the following statement:

```
CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file connected to unit J and deletes it. If an error occurs, control is transferred to the statement labeled 99.

Consider the following statement:

```
CLOSE (UNIT=1, STATUS='PRINT')
```

This statement closes the file on unit 1 and submits it for printing.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On the READONLY specifier, see Section 12.6.24.

12.3. DELETE Statement

The DELETE statement deletes a record from a relative or indexed organization file. It takes one of the following forms:

```
DELETE ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var])

DELETE ([UNIT=]io-unit [,REC=r] [,ERR=label] [,IOSTAT=i-var])
```

`io-unit`

Is an external unit specifier.

`r`

Is a scalar numeric expression indicating the record number to be deleted.

`label`

Is the label of the branch target statement that receives control if an error occurs.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

In files with keyed access, the DELETE statement deletes the current record. The current record is the last record that is accessed by a READ statement on the specified external unit.

In files with direct access, the DELETE statement deletes the direct access record specified by *r*. If REC=r is omitted, the current record is deleted. When the direct access record is deleted, any associated variable is set to the next record number.

The DELETE statement logically removes the appropriate record from the specified file by locating the record and marking it as a deleted record. It then frees the position formerly occupied by the deleted record so that a new record can be written into that position.

Examples

The following statement deletes the fifth record in the file connected to I/O unit 10:

```
DELETE (10, REC=5)
```

In the next example, the current record is deleted from the file connected to I/O unit 11:

```
DELETE (11)
```

For More Information:

- On an alternative form for the DELETE statement, see Section B.9.
- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On the REC control specifier, see Section 10.2.1.4.

12.4. ENDFILE Statement

For sequential files, the ENDFILE statement writes an end-of-file record to the file and positions the file after this record (the terminal point). For direct access files, the ENDFILE statement truncates the file after the current record.

An ENDFILE statement takes one of the following forms:

```
ENDFILE ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var])
ENDFILE io-unit
```

`io-unit`

Is an external unit specifier.

`label`

Is the label of the branch target statement that receives control if an error occurs.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

If the unit specified in the ENDFILE statement is not open, the default file is opened for unformatted output.

An end-of-file record can be written only to files with sequential organization that are accessed as formatted-sequential or unformatted-segmented sequential files.

An ENDFILE statement performed on a direct access file always truncates the file.

An ENDFILE statement must not be issued for a file that is open for keyed access.

An end-of-file record written to a file on magnetic tape is not the same as a tape mark.

End-of-file records should not be written in files that are read by programs written in a language other than Fortran, because other languages do not support the embedded end-of-file concept.

Examples

The following statement writes an end-of-file record to I/O unit 2:

```
ENDFILE 2
```

Suppose the following statement is specified:

```
ENDFILE (UNIT=9, IOSTAT=IOS, ERR=10)
```

An end-of-file record is written to the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On record position, advancement, and transfer, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.5. INQUIRE Statement

The INQUIRE statement returns information on the status of specified properties of a file or logical unit. It takes one of the following forms:

```
INQUIRE (FILE=name [,ERR=label] [,IOSTAT=i-var] [,DEFAULTFILE=def], slist)

INQUIRE ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var], slist)

INQUIRE (IOLENGTH=len) out-item-list
```

`name`

Is a scalar default character expression specifying the name of the file for inquiry.

`label`

Is the label of the branch target statement that receives control if an error occurs.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

`def`

Is a scalar default character expression specifying a default file name specification string. (For more information on the DEFAULTFILE specifier, see the Section 12.6.10).

slist

Is one or more inquiry specifiers. Each specifier can appear only once. (The inquiry specifiers are described individually in the following sections).

io-unit

Is an external unit specifier.

The unit does not have to exist, nor does it need to be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

len

Is a scalar integer variable that is assigned a value corresponding to the length of an unformatted, direct-access record resulting from the use of the *out-item-list* in a WRITE statement.

The value is suitable to use as a RECL specifier value in an OPEN statement that connects a file for unformatted, direct access.

The unit of the value is 4-byte longwords, by default. However, if you specify the compiler option /ASSUME=BYTERECL, the unit is bytes.

out-item-list

Is a list of one or more output items (see Section 10.2.2).

Rules and Behavior

The control specifiers ([UNIT=]io-unit, ERR=label, and IOSTAT=i-var) and inquiry specifiers can appear anywhere within the parentheses following INQUIRE. However, if the UNIT specifier is omitted, the *io-unit* must appear first in the list.

An INQUIRE statement can be executed before, during, or after a file is connected to a unit. The specifier values returned are those that are current when the INQUIRE statement executes.

To get file characteristics, specify the INQUIRE statement after opening the file.

Examples

The following are examples of INQUIRE statements:

```
INQUIRE (FILE='FILE_B', EXIST=EXT)
INQUIRE (4, FORM=FM, IOSTAT=IOS, ERR=20)
INQUIRE (IOLENGTH=LEN) A, B
```

In the last statement, you can use the length returned in LEN as the value for the RECL specifier in an OPEN statement that connects a file for unformatted direct access. If you have already specified a value for RECL, you can check LEN to verify that A and B are less than or equal to the record length you specified.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.

- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On the RECL specifier in OPEN statements, see Section 12.6.25.
- On the FILE specifier in OPEN statements, see Section 12.6.14.
- On the DEFAULTFILE specifier in OPEN statements, see Section 12.6.10.

12.5.1. ACCESS Specifier

The ACCESS specifier asks how a file is connected. It takes the following form:

```
ACCESS = acc
```

`acc`

Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is connected for sequential access
'DIRECT'	If the file is connected for direct access
'KEYED'	If the file is connected for keyed access
'UNDEFINED'	If the file is not connected

12.5.2. ACTION Specifier

The ACTION specifier asks which I/O operations are allowed for a file. It takes the following form:

```
ACTION = act
```

`act`

Is a scalar default character variable that is assigned one of the following values:

'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
'UNDEFINED'	If the file is not connected

12.5.3. BLANK Specifier

The BLANK specifier asks what type of blank control is in effect for a file. It takes the following form:

```
BLANK = blnk
```

`blnk`

Is a scalar default character variable that is assigned one of the following values:

'NULL'	If null blank control is in effect for the file

'ZERO'	If zero blank control is in effect for the file
'UNDEFINED'	If the file is not connected, or it is not connected for formatted data transfer

12.5.4. BLOCKSIZE Specifier

The BLOCKSIZE specifier asks about the I/O buffer size. It takes the following form:

```
BLOCKSIZE = bks
```

bks

Is a scalar integer variable.

The *bks* is assigned the current size of the I/O buffer. If the unit or file is not connected, the value assigned is zero.

12.5.5. BUFFERED Specifier

The BUFFERED specifier asks whether run-time buffering is in effect. It takes the following form:

```
BUFFERED = bf
```

bf

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file or unit is connected and buffering is in effect.
'NO'	If the file or unit is connected, but buffering is not in effect.
'UNKNOWN'	If the file or unit is not connected.

12.5.6. CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier asks what type of carriage control is in effect for a file. It takes the following form:

```
CARRIAGECONTROL = cc
```

cc

Is a scalar default character variable that is assigned one of the following values:

'FORTRAN'	If the file is connected with Fortran carriage control in effect
'LIST'	If the file is connected with implied carriage control in effect
'NONE'	If the file is connected with no carriage control in effect
'UNKNOWN'	If the file is not connected, or if it has an unsupported carriage control type

12.5.7. CONVERT Specifier

The CONVERT specifier asks what type of data conversion is in effect for a file. It takes the following form:

```
CONVERT = fm
```

fm

Is a scalar default character variable that is assigned one of the following values:

'LITTLE_ENDIAN'	If the file is connected with little endian integer and IEEE floating-point data conversion in effect
'BIG_ENDIAN'	If the file is connected with big endian integer and IEEE floating-point data conversion in effect
'CRAY'	If the file is connected with big endian integer and CRAY ® floating-point data conversion in effect
'FDX'	If the file is connected with little endian integer and VAX F_floating, D_floating, and IEEE X_floating data conversion in effect
'FGX'	If the file is connected with little endian integer and VAX F_floating, G_floating, and IEEE X_floating data conversion in effect
'IBM'	If the file is connected with big endian integer and IBM ® System \370 floating-point data conversion in effect
'VAXD'	If the file is connected with little endian integer and VAX F_floating, D_floating, and H_floating in effect
'VAXG'	If the file is connected with little endian integer and VAX F_floating, G_floating, and H_floating in effect
'NATIVE'	If the file is connected with no data conversion in effect
'UNKNOWN'	If the file or unit is not connected for unformatted data transfer

12.5.8. DELIM Specifier

The DELIM specifier asks how character constants are delimited in list-directed and namelist output. It takes the following form:

```
DELIM = del
```

del

Is a scalar default character variable that is assigned one of the following values:

'APOSTROPHE'	If apostrophes are used to delimit character constants in list-directed and namelist output
'QUOTE'	If quotation marks are used to delimit character constants in list-directed and namelist output
'NONE'	If no delimiters are used
'UNDEFINED'	If the file is not connected, or is not connected for formatted data transfer

12.5.9. DIRECT Specifier

The DIRECT specifier asks whether a file is connected for direct access. It takes the following form:

```
DIRECT = dir
```

dir

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for direct access
'NO'	If the file is not connected for direct access
'UNKNOWN'	If the file is not connected

12.5.10. EXIST Specifier

The EXIST specifier asks whether a file exists and can be opened. It takes the following form:

```
EXIST = ex
```

ex

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file exists and can be opened, or if the specified unit exists
.FALSE.	If the specified file or unit does not exist or if the file exists but cannot be opened

The unit exists if it is a number in the range allowed by the processor.

12.5.11. FORM Specifier

The FORM specifier asks whether a file is connected for formatted or unformatted data transfer. It takes the following form:

```
FORM = fm
```

fm

Is a scalar default character variable that is assigned one of the following values:

'FORMATTED'	If the file is connected for formatted data transfer
'UNFORMATTED'	If the file is connected for unformatted data transfer
'UNDEFINED'	If the file is not connected

12.5.12. FORMATTED Specifier

The FORMATTED specifier asks whether a file is connected for formatted data transfer. It takes the following form:

```
FORMATTED = fmt
```

fmt

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for formatted data transfer

'NO'	If the file is not connected for formatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for formatted data transfer

12.5.13. KEYED Specifier

The KEYED specifier asks whether a file is connected for keyed access. It takes the following form:

```
KEYED = kyd
```

kyd

Is a scalar default character variable that is assigned one of the following values:

'YES'	If keyed access is allowed for the indexed file
'NO'	If keyed access is not allowed
'UNKNOWN'	If the processor cannot determine whether keyed access is allowed

12.5.14. NAME Specifier

The NAME specifier returns the name of a file. It takes the following form:

```
NAME = nme
```

nme

Is a scalar default character variable that is assigned the name of the file to which the unit is connected. If the file does not have a name, *nme* is undefined.

The value assigned to *nme* is not necessarily the same as the value given in the FILE specifier. For example, the value that the processor returns may be qualified by a directory name or a version number.

However, the value that is assigned is always valid for use with the FILE specifier in an OPEN statement, unless the value has been truncated in a way that makes it unacceptable. (Values are truncated if the declaration of *nme* is too small to contain the entire value).

Note

The FILE and NAME specifiers are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

For More Information:

On the maximum possible size of file specifications, see the *OpenVMS Record Management Services Reference Manual*.

12.5.15. NAMED Specifier

The NAMED specifier asks whether a file is named. It takes the following form:

```
NAMED = nmd
```

nmd

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the file has a name
.FALSE.	If the file does not have a name

12.5.16. NEXTREC Specifier

The NEXTREC specifier asks where the next record can be read or written in a file connected for direct access. It takes the following form:

```
NEXTREC = nr
```

nr

Is a scalar integer variable that is assigned a value as follows:

- If the file is connected for direct access and a record (r) was previously read or written, the value assigned is r + 1.
- If no record has been read or written, the value assigned is 1.
- If the file is not connected for direct access, or if the file position cannot be determined because of an error condition, the value assigned is zero.
- If the file is connected for direct access and a REWIND has been performed on the file, the value assigned is 1.

12.5.17. NUMBER Specifier

The NUMBER specifier asks the number of the unit connected to a file. It takes the following form:

```
NUMBER = num
```

num

Is an scalar integer variable.

The *num* is assigned the number of the unit currently connected to the specified file. If there is no unit connected to the file, *num* is not defined.

12.5.18. OPENED Specifier

The OPENED specifier asks whether a file is connected. It takes the following form:

```
OPENED = od
```

od

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file or unit is connected

.FALSE.	If the specified file or unit is not connected

12.5.19. ORGANIZATION Specifier

The ORGANIZATION specifier asks how the file is organized. It takes the following form:

```
ORGANIZATION = org
```

org

Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is a sequential file
'RELATIVE'	If the file is a relative file
'INDEXED'	If the file is an indexed file
'UNKNOWN'	If the processor cannot determine the file's organization

12.5.20. PAD Specifier

The PAD specifier asks whether blank padding was specified for the file. It takes the following form:

```
PAD = pd
```

pd

Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit was connected with PAD='NO'
'YES'	If the file or unit is not connected, or it was connected with PAD='YES'

12.5.21. POSITION Specifier

The POSITION specifier asks the position of the file. It takes the following form:

```
POSITION = pos
```

pos

Is a scalar default character variable that is assigned one of the following values:

'REWIND'	If the file is connected with its position at its initial point
'APPEND'	If the file is connected with its position at its terminal point (or before its end-of-file record, if any)
'ASIS'	If the file is connected without changing its position
'UNDEFINED'	If the file is not connected, or is connected for direct access data transfer and a REWIND statement has not been performed on the unit.

For More Information:

On record position, advancement, and transfer, see *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.5.22. READ Specifier

The READ specifier asks whether a file can be read. It takes the following form:

```
READ = rd
```

rd

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be read
'NO'	If the file cannot be read
'UNKNOWN'	If the processor cannot determine whether the file can be read

12.5.23. READWRITE Specifier

The READWRITE specifier asks whether a file can be both read and written to. It takes the following form:

```
READWRITE = rdwr
```

rdwr

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be both read and written to
'NO'	If the file cannot be both read and written to
'UNKNOWN'	If the processor cannot determine whether the file can be both read and written to

12.5.24. RECL Specifier

The RECL specifier asks the maximum record length for a file. It takes the following form:

```
RECL = rcl
```

rcl

Is a scalar integer variable that is assigned a value as follows:

- If the file or unit is connected, the value assigned is the maximum record length allowed.
- If the file is not connected, the value assigned is the maximum record length allowed in the file. However, if the maximum record length is zero, the value assigned is the length of the longest record in the file.

  If inquiring about a file that has no maximum record size, see Section 12.6.25.
- If the file is segmented, the value assigned is the longest segment length in the file.
- If the file does not exist, the value assigned is zero.

The assigned value is expressed in 4-byte units if a file is currently (or was previously) connected for unformatted data transfer; otherwise, the value is expressed in bytes.

12.5.25. RECORDTYPE Specifier

The RECORDTYPE specifier asks which type of records are in a file. It takes the following form:

```
RECORDTYPE = rtype
```

rtype

Is a scalar default character variable that is assigned one of the following values:

'FIXED'	If the file is connected for fixed-length records
'VARIABLE'	If the file is connected for variable-length records
'SEGMENTED'	If the file is connected for unformatted sequential data transfer using segmented records
'STREAM'	If the file's records are terminated with a carriage return and line feed
'STREAM_CR'	If the file's records are terminated with only a carriage return
'STREAM_LF'	If the file's records are terminated with only a line feed
'UNKNOWN'	If the processor cannot determine the record type

12.5.26. SEQUENTIAL Specifier

The SEQUENTIAL specifier asks whether a file is connected for sequential access. It takes the following form:

```
SEQUENTIAL = seq
```

seq

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for sequential access
'NO'	If the file is not connected for sequential access
'UNKNOWN'	If the processor cannot determine whether the file is connected for sequential access

12.5.27. UNFORMATTED Specifier

The UNFORMATTED specifier asks whether a file is connected for unformatted data transfer. It takes the following form:

```
UNFORMATTED = unf
```

unf

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for unformatted data transfer
'NO'	If the file is not connected for unformatted data transfer

'UNKNOWN'	If the processor cannot determine whether the file is connected for unformatted data transfer

12.5.28. WRITE Specifier

The WRITE specifier asks whether a file can be written to. It takes the following form:

```
WRITE = wr
```

wr

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be written to
'NO'	If the file cannot be written to
'UNKNOWN'	If the processor cannot determine whether the file can be written to

12.6. OPEN Statement

The OPEN statement connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection.

The OPEN statement takes the following form:

```
OPEN ([UNIT=]io-unit [,FILE=name] [,ERR=label] [,IOSTAT=i-var], slist)
```

io-unit

Is an external unit specifier.

name

Is a character or numeric expression specifying the name of the file to be connected. For more information, see Section 12.6.14.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

slist

Is one or more OPEN specifiers in the form *specifier*=*value* or *specifier*. Each specifier can appear only once.

The OPEN specifiers and their acceptable values are summarized in Table 12.1.

The OPEN specifiers are described individually in the following sections. The control specifiers that can be specified in an OPEN statement (UNIT, ERR, and IOSTAT) are discussed in Section 10.2.1.

Table 12.1. OPEN Statement Specifiers and Values

Specifier	Values	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'	Access mode	'SEQUENTIAL'
ACTION	'READ' 'WRITE' 'READWRITE'	File access	'READWRITE'
ASSOCIATEVARIABLE	var	Next direct access record	No default
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	n_expr	Physical block size	System default
BUFFERCOUNT	n_expr	Number of I/O buffers	System default
BUFFERED	'YES' 'NO'	Buffering for WRITE operations	'NO'
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	Formatted: 'FORTRAN' Unformatted: 'NONE'
CONVERT	'LITTLE_ENDIAN' 'BIG_ENDIAN' 'CRAY' 'FDX' 'FGX' 'IBM' 'VAXD' 'VAXG' 'NATIVE'	Numeric format specification	'NATIVE'
DEFAULTFILE	c_expr	Default file specification	Current working directory
DELIM	'APOSTROPHE' 'QUOTE' 'NONE'	Delimiter for character constants	'NONE'
DISPOSE (or DISP)	'KEEP' or 'SAVE' 'DELETE'	File disposition at close	'KEEP'

Key to Values

c_expr: A scalar default character expression
dr: A direction, `ASCENDING` or `DESCENDING`
dt: A data type, `INTEGER` or `CHARACTER`
e1: The first byte position of a key
e2: The last byte position of a key
func: An external function
label: A statement label
n_expr: A scalar numeric expression
var: A scalar integer variable

Specifier	Values	Function	Default
	'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'		
ERR	label	Error transfer control	No default
EXTENDSIZE	n_expr	File allocation increment	Volume or system default
FILE (or NAME)	c_expr	File specification (file name)	FORnnn.DAT[1]
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS setting
INITIALSIZE	n_expr	File allocation	No default
IOSTAT	var	I/O status	No default
KEY	(e1:e2[:dt[:dr]],...)	Key field definitions	CHARACTER ASCENDING
MAXREC	n_expr	Direct access record limit	No limit
NOSPANBLOCKS	No value	Records do not span blocks	No default
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'	File organization	'SEQUENTIAL'
PAD	'YES' 'NO'	Record padding	'YES'
POSITION	'ASIS' 'REWIND' 'APPEND'	File positioning	'ASIS'
READONLY	No value	Write protection	No default
RECL (or RECORDSIZE)	n_expr	Record length	Depends on RECORDTYPE, ORGANIZATION, and FORM settings

Key to Values

c_expr: A scalar default character expression
dr: A direction, `ASCENDING` or `DESCENDING`
dt: A data type, `INTEGER` or `CHARACTER`
e1: The first byte position of a key
e2: The last byte position of a key
func: An external function
label: A statement label
n_expr: A scalar numeric expression
var: A scalar integer variable

Specifier	Values	Function	Default
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'	Record type	Depends on ORGANIZATION, ACCESS, and FORM settings
SHARED	No value	File sharing allowed	No default[2]
STATUS (or TYPE)	'OLD' 'NEW' 'SCRATCH' 'REPLACE' 'UNKNOWN'	File status at open	'UNKNOWN'[3]
UNIT	n_expr	Logical unit number	No default; an io-unit must be specified
USEROPEN	func	User program option	No default

Key to Values

c_expr: A scalar default character expression
dr: A direction, `ASCENDING` or `DESCENDING`
dt: A data type, `INTEGER` or `CHARACTER`
e1: The first byte position of a key
e2: The last byte position of a key
func: An external function
label: A statement label
n_expr: A scalar numeric expression
var: A scalar integer variable

[1] nnn is the unit number (with leading zeros, if necessary).

[2] For information on file sharing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

[3] The default differs under certain conditions (see Section 12.6.29).

Rules and Behavior

The control specifiers ([UNIT=]io-unit, ERR=label, and IOSTAT=i-var) and OPEN specifiers can appear anywhere within the parentheses following OPEN. However, if the UNIT specifier is omitted, the *io-unit* must appear first in the list.

Specifier values that are scalar numeric expressions can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

Only one unit at a time can be connected to a file, but multiple OPENs can be performed on the same unit. If an OPEN statement is executed for a unit that already exists, the following occurs:

- If FILE is not specified, or FILE specifies the same file name that appeared in a previous OPEN statement, the current file remains connected.

  If the file names are the same, the values for the BLANK, CONVERT, CARRIAGECONTROL, DELIM, DISPOSE, ERR, IOSTAT, and PAD specifiers can be changed. Other OPEN specifier values cannot be changed, and the file position is unaffected.

- If FILE specifies a different file name, the previous file is closed and the new file is connected to the unit.

The ERR and IOSTAT specifiers from any previously executed OPEN statement have no effect on any currently executing OPEN statement. If an error occurs, no file is opened or created.

Secondary operating system messages do not display when IOSTAT is specified. To display these messages, remove IOSTAT or use a platform-specific method. (For more information, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/]).

Examples

You can specify character values at run time by substituting a character expression for a specifier value in the OPEN statement. The character value can contain trailing blanks but not leading or embedded blanks; for example:

```
CHARACTER*7 QUAL /' '/
...
IF (exp) QUAL = '/DELETE'
  OPEN (UNIT=1, STATUS='NEW', DISP='SUBMIT'//QUAL)
```

The following statement creates a new sequential formatted file on unit 1 with the default file name FOR001.DAT:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The following statement creates a 50-block direct access file for temporary storage. The file is deleted at program termination.

```
OPEN (UNIT=3, STATUS='SCRATCH', ACCESS='DIRECT',              &
     INITIALSIZE=50, RECL=64)
```

The following statement creates a file on magnetic tape with a large block size for efficient processing:

```
 OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', BLOCKSIZE=8192,
1     STATUS='NEW', ERR=14, RECL=1024,
1     RECORDTYPE='FIXED')
```

The following statement opens the file (created in the previous example) for input:

```
 OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', READONLY,
1     STATUS='OLD', RECL=1024, RECORDTYPE='FIXED',
1     BLOCKSIZE=8192)
```

The following statement uses the file name supplied by the user and the default file specification supplied by the DEFAULTFILE specifier to define the file specification for an existing file:

```
 TYPE *, 'ENTER NAME OF DOCUMENT'
 ACCEPT *, DOC
 OPEN (UNIT=1, FILE=DOC, DEFAULTFILE='[ARCHIVE].TXT',
1     STATUS='OLD')
```

For More Information:

- On Fortran IOSTAT errors, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On using the INQUIRE statement to get file attributes of existing files, see Section 12.5.
- On OPEN statements and file connection, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.1. ACCESS Specifier

The ACCESS specifier indicates the access method for the connection of the file. It takes the following form:

```
ACCESS = acc
```

acc

Is a scalar default character expression that evaluates to one of the following values:

'DIRECT'	Indicates direct access.
'SEQUENTIAL'	Indicates sequential access.
'KEYED'	Indicates keyed access.
'APPEND'	Indicates sequential access, but the file is positioned at the end-of-file record.

The default is 'SEQUENTIAL'.

12.6.2. ACTION Specifier

The ACTION specifier indicates the allowed I/O operations for the file connection. It takes the following form:

```
ACTION = act
```

act

Is a scalar default character expression that evaluates to one of the following values:

'READ'	Indicates that only READ statements can refer to this connection.
'WRITE'	Indicates that only WRITE, DELETE, and ENDFILE statements can refer to this connection.
'READWRITE'	Indicates that READ, WRITE, DELETE, and ENDFILE statements can refer to this connection.

The default is 'READWRITE'.

12.6.3. ASSOCIATEVARIABLE Specifier

The ASSOCIATEVARIABLE specifier indicates a variable that is updated after each direct access I/O operation, to reflect the record number of the next sequential record in the file. It takes the following form:

```
ASSOCIATEVARIABLE = asv
```

asv

Is a scalar integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

Direct access READs, direct access WRITEs, and the FIND, DELETE, and REWRITE statements can affect the value of *asv*.

This specifier is valid only for direct access; it is ignored for other access modes.

12.6.4. BLANK Specifier

The BLANK specifier indicates how blanks are interpreted in a file. It takes the following form:

```
BLANK = blnk
```

blnk

Is a scalar default character expression that evaluates to one of the following values:

'NULL'	Indicates all blanks are ignored, except for an all-blank field (which has a value of zero).
'ZERO'	Indicates all blanks (other than leading blanks) are treated as zeros.

The default is 'NULL' (for explicitly OPENed files, preconnected files, and internal files). If you specify compiler option /NOF77 (or OPTIONS /NOF77), the default is 'ZERO'.

If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks.

For More Information:

On the BN and BZ edit descriptors, see Section 11.4.4.

12.6.5. BLOCKSIZE Specifier

The BLOCKSIZE specifier specifies the physical I/O transfer size for the file. It takes the following form:

```
BLOCKSIZE = bks
```

bks

Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

For magnetic tape files, the value of *bks* specifies the physical record size in the range 18 to 32767 bytes. The default value is 2048 bytes.

For sequential disk files, the value of *bks* is rounded up to an integral number of 512-byte blocks and used to specify multiblock transfers. The number of blocks transferred can be 1 to 127; it is determined by RMS defaults.

For relative and indexed files, the value of *bks* is rounded up to an integral number of 512-byte blocks, and is used to specify the RMS bucket size in the range 1 to 63 blocks. The default is the smallest value capable of holding a single record.

For More Information:

- On setting RMS defaults, see the SET RMS_DEFAULT command in the *VSI OpenVMS DCL Dictionary*.
- On tuning information, see the *VSI OpenVMS Guide to OpenVMS File Applications*.

12.6.6. BUFFERCOUNT Specifier

The BUFFERCOUNT specifier indicates the number of buffers to be associated with the unit for multibuffered I/O. It takes the following form:

```
BUFFERCOUNT = bc
```

bc

Is a scalar numeric expression in the range 1 through 127. If necessary, the value is converted to integer data type before use.

The BLOCKSIZE specifier determines the size of each buffer. For example, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144.

If you do not specify BUFFERCOUNT or you specify zero for *bc*, the process or system default is assumed.

For More Information:

- On setting RMS defaults, see the *VSI OpenVMS DCL Dictionary*.
- On the BLOCKSIZE specifier, see Section 12.6.5.

12.6.7. BUFFERED Specifier

The BUFFERED specifier indicates run-time library behavior following WRITE operations. It takes the following form:

```
BUFFERED = bf
```

bf

Is a scalar default character expression that evaluates to one of the following values:

'NO'	Requests that the run-time library send output data to the file system after each WRITE operation.
'YES'	Requests that the run-time library accumulate output data in its internal buffer, possibly across several WRITE operations, before the data is sent to the file system. Buffering may improve run-time performance for output-intensive applications.

The default is 'NO'.

BUFFERED has no effect. The operating system automatically performs buffering, which can be affected by the values of the BUFFERCOUNT and BUFFERSIZE keywords when the file is opened.

12.6.8. CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier indicates the type of carriage control used when a file is printed. It takes the following form:

```
CARRIAGECONTROL = cc
```

cc

Is a scalar default character expression that evaluates to one of the following values:

'FORTRAN'	Indicates normal Fortran interpretation of the first character.
'LIST'	Indicates one line feed between records.
'NONE'	Indicates no carriage control processing.

The default for formatted files is 'FORTRAN'. The default for unformatted files, is 'NONE'.

12.6.9. CONVERT Specifier

The CONVERT specifier indicates a nonnative numeric format for unformatted data. It takes the following form:

```
CONVERT = fm
```

fm

Is a scalar default character expression that evaluates to one of the following values:

'LITTLE_ENDIAN'[1]	Little endian integer data[2] and IEEE floating-point data.[3]
'BIG_ENDIAN'[1]	Big endian integer data[2] and IEEE floating-point data.[3]
'CRAY'	Big endian integer data[2] and CRAY floating-point data of size REAL(8) or COMPLEX(8).
'FDX'	Little endian integer data[2] and VAX floating-point data of format F_floating for REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'FGX'	Little endian integer data[2] and VAX floating-point data of format F_floating for REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'IBM'	Big endian integer data[2] and IBM System \370 floating-point data of size REAL(4) or COMPLEX(4) (IBM short 4), and size REAL(8) or COMPLEX(8) (IBM long 8).
'VAXD'	Little endian integer data[2] and VAX floating-point data of format F_floating for size REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'VAXG'	Little endian integer data[2] and VAX floating-point data of format F_floating for size REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).

'NATIVE'	No data conversion. This is the default.

[1]INTEGER(1) data is the same for little endian and big endian.

[2]Of the appropriate size: INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)

[3]Of the appropriate size and type: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), or COMPLEX(16)

You can use CONVERT to specify multiple formats in a single program, usually one format for each specified unit number.

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message appears.

There are other ways to specify numeric format for unformatted files: you can specify an OpenVMS logical name, the command line qualifier /CONVERT, or OPTIONS/CONVERT. The following shows the order of precedence:

Method Used	**Precedence**
OpenVMS logical name	Highest
OPEN (CONVERT=)	.
OPTIONS/CONVERT	.
The /CONVERT qualifier	Lowest

The /CONVERT qualifier and OPTIONS/CONVERT affect all unit numbers used by the program, while logical names and OPEN (CONVERT=) affect specific unit numbers.

The following example shows how to code the OPEN statement to read unformatted CRAY numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20:

```
 OPEN (CONVERT='CRAY', FILE='graph3.dat', FORM='UNFORMATTED',
1     UNIT=15)
      .
      .
      .
 OPEN (FILE='graph3_native.dat', FORM='UNFORMATTED', UNIT=20)
```

For More Information:

- On transporting data between VSI Fortran platforms, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On supported ranges for data types, see Chapter 3 and *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On using OpenVMS logical names to specify CONVERT options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On qualifiers, in general, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.10. DEFAULTFILE Specifier

The DEFAULTFILE specifier indicates a default file specification string. It takes the following form:

```
DEFAULTFILE = def
```

def

Is a character expression indicating a default file specification string.

This specifier can supply a value to the RMS default file specification string for the missing components of a file specification. If you omit the DEFAULTFILE specifier, VSI Fortran uses the default value “FOR *nnn*.DAT”, where *nnn* is the unit number with leading zeros.

The default file specification string is used primarily when accepting file specifications interactively. Complete file specifications known to a user program normally appear in the FILE specifier.

You can indicate default values for any one of the following file-specification components:

- Node
- Device
- Directory
- File name
- File type
- File version number

If you indicate values for any of these components in the FILE specifier, they override any values indicated in the DEFAULTFILE specifier.

For More Information:

On specifying file-specification components, see the OpenVMS Record Management Services Reference Manual.

12.6.11. DELIM Specifier

The DELIM specifier indicates what characters (if any) are used to delimit character constants in list-directed and namelist output. It takes the following form:

```
DELIM = del
```

del

Is a scalar default character expression that evaluates to one of the following values:

'APOSTROPHE'	Indicates apostrophes delimit character constants. All internal apostrophes are doubled.
'QUOTE'	Indicates quotation marks delimit character constants. All internal quotation marks are doubled.
'NONE'	Indicates character constants have no delimiters. No internal apostrophes or quotation marks are doubled.

The default is 'NONE'.

The DELIM specifier is only allowed for files connected for formatted data transfer; it is ignored during input.

12.6.12. DISPOSE Specifier

The DISPOSE (or DISP) specifier indicates the status of the file after the unit is closed. It takes one of the following forms:

```
{DISPOSE = dis | DISP = dis}
```

dis

Is a scalar default character expression that evaluates to one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.
'PRINT'[1]	Submits the file to the system line printer spooler and retains it.
'PRINT/DELETE'[1]	Submits the file to the system line printer spooler and then deletes it.
'SUBMIT'	Submits the file to the batch job queue and then retains it.
'SUBMIT/ DELETE'	Submits the file to the batch job queue and then deletes it.

[1]Use only on sequential files.

A read-only file cannot be deleted.

The default is 'DELETE' for scratch files; a scratch file cannot be saved, printed, or submitted. For all other files, the default is 'KEEP'.

12.6.13. EXTENDSIZE Specifier

The EXTENDSIZE specifier indicates the number of blocks by which to extend a disk file (extent) when additional storage space is needed. It takes the following form:

```
EXTENDSIZE = es
```

es

Is a scalar numeric expression.

If you do not specify EXTENDSIZE or if you specify zero, the process or system default for the device is used.

For More Information:

On the relationship between the EXTENDSIZE specifier and the INITIALSIZE specifier, see Section 12.6.16.

12.6.14. FILE Specifier

The FILE specifier indicates the name of the file to be connected to the unit. It takes the following form:

```
FILE = name
```

name

Is a character or numeric expression.

The *name* can be any specification allowed by the operating system.

Any trailing blanks in the name are ignored.

If the following conditions occur:

- FILE is omitted
- The unit is not connected to a file
- STATUS='SCRATCH' is not specified

then VSI Fortran generates a file name in the form FOR nnn.DAT, where nnn is the logical unit number (with leading zeros, if necessary).

If the file name is stored in a numeric scalar or array, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character scalar or array, it must not contain a zero byte.

For More Information:

- On default file name conventions, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On allowable file specifications, see the appropriate manual in your system documentation set.

12.6.15. FORM Specifier

The FORM specifier indicates whether the file is being connected for formatted or unformatted data transfer. It takes the following form:

```
FORM = fm
```

fm

Is a scalar default character expression that evaluates to one of the following values:

'FORMATTED'	Indicates formatted data transfer
'UNFORMATTED'	Indicates unformatted data transfer

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct and keyed access files.

12.6.16. INITIALSIZE Specifier

The INITIALSIZE specifier indicates the number of blocks in the initial storage allocation (extent) for a disk file. This information is used by the EXTENDSIZE specifier, which indicates the number of blocks by which a disk file is extended each time more space is needed for a file. The INITIALSIZE specifier takes the following form:

```
INITIALSIZE = insz
```

insz

Is a scalar numeric expression.

If you do not specify INITIALSIZE or if you specify zero, no initial allocation is made. The system attempts to allocate contiguous space for INITIALSIZE, but noncontiguous space is allocated if there is not enough contiguous space available.

INITIALSIZE is effective only at the time the file is created. If EXTENDSIZE is specified when the file is created, the value specified is the default value used to allocate additional storage for the file.

If you specify EXTENDSIZE when you open an existing file, the value you specify supersedes any EXTENDSIZE value specified when the file was created, and remains in effect until you close the file. Unless specifically overridden, the default EXTENDSIZE value is in effect on subsequent openings of the file.

12.6.17. KEY Specifier

The KEY specifier defines the access keys for records in an indexed file. It takes the following form:

```
KEY = (kspec [,kspec]...)
```

kspec

Takes the following form:

```
e1:e2 [:dt[:dr]]
```

e1

Is the first byte position of the key.

e2

Is the last byte position of the key.

dt

Is the data type of the key: INTEGER or CHARACTER.

dr

Is the direction of the key: ASCENDING or DESCENDING.

The defaults are CHARACTER and ASCENDING.

The key starts at position *e1* in a record and has a length of e2-e1+1. The values of *e1* and *e2* must cause the following calculations to be true:

```
1 .LE. (e1) .AND. (e1) .LE. (e2) .AND. (e2) .LE. record-length
1 .LE. (e2-e1+1) .AND. (e2-e1+1) .LE. 255
```

If the key type is INTEGER, the key length must be either 2 or 4.

Defining Primary and Alternate Keys

You must define at least one key in an indexed file. This is the primary key (the default key). It usually has a unique value for each record.

You can also define alternate keys. RMS allows up to 254 alternate keys.

If a file requires more keys than the OPEN statement limit, you must create the file using another language or the File Definition Language (FDL).

Specifying and Referencing Keys

You must use the KEY specifier when creating an indexed file. However, you do not have to respecify it when opening an existing file, because key attributes are permanent aspects of the file. These attributes include key definitions and reference numbers for subsequent I/O operations.

However, if you use the KEY specifier for an existing file, your specification must be identical to the established key attributes.

Subsequent I/O operations use a reference number, called the key-of-reference number, to identify a particular key. You do not specify this number; it is determined by the key's position in the specification list: the primary key is key-of-reference number 0; the first alternate key is key-of-reference number 1, and so forth.

For More Information:

On the FDL, see the OpenVMS Record Management Services Reference Manual.

12.6.18. MAXREC Specifier

The MAXREC specifier indicates the maximum number of records that can be transferred from or to a direct access file while the file is connected. It takes the following form:

```
MAXREC = mr
```

mr

Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

The default is the maximum allowed (2**32–1).

12.6.19. NAME Specifier

NAME is a nonstandard synonym for FILE (see Section 12.6.14).

12.6.20. NOSPANBLOCKS Specifier

The NOSPANBLOCKS specifier indicates that records are not to cross disk block boundaries. It takes the following form:

```
NOSPANBLOCKS
```

This specifier causes an error to occur if any record exceeds the size of a physical block.

12.6.21. ORGANIZATION Specifier

The ORGANIZATION specifier indicates the internal organization of the file. It takes the following form:

```
ORGANIZATION = org
```

org

Is a scalar default character expression that evaluates to one of the following values:

'SEQUENTIAL'	Indicates a sequential file.
'RELATIVE'	Indicates a relative file.
'INDEXED'	Indicates an indexed file.

The default is 'SEQUENTIAL'. However, if you omit the ORGANIZATION specifier when you open an existing file, the organization already specified in that file is used. If you specify ORGANIZATION for an existing file, *org* must have the same value as that of the existing file.

12.6.22. PAD Specifier

The PAD specifier indicates whether a formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.

The PAD specifier takes the following form:

```
PAD = pd
```

pd

Is a scalar default character expression that evaluates to one of the following values:

'YES'	Indicates the record will be padded with blanks when necessary.
'NO'	Indicates the record will not be padded with blanks. The input record must contain the data required by the input list and format specification.

The default is 'YES'.

This behavior is different from FORTRAN 77, which never pads short records with blanks. For example, consider the following:

```
READ (5,'(I5)') J
```

If you enter 123 followed by a carriage return, FORTRAN 77 turns the I5 into an I3 and J is assigned 123.

However, VSI Fortran pads the 123 with 2 blanks unless you explicitly open the unit with PAD='NO'.

You can override blank padding by explicitly specifying the BN edit descriptor.

The PAD specifier is ignored during output.

12.6.23. POSITION Specifier

The POSITION specifier indicates the position of a file connected for sequential access. It takes the following form:

```
POSITION = pos
```

pos

Is a scalar default character expression that evaluates to one of the following values:

'ASIS'	Indicates the file position is unchanged if the file exists and is already connected. The position is unspecified if the file exists but is not connected.
'REWIND'	Indicates the file is positioned at its initial point.
'APPEND'	Indicates the file is positioned at its terminal point (or before its end-of-file record, if any).

The default is 'ASIS'.

A new file (whether specified as new explicitly or by default) is always positioned at its initial point.

For More Information:

On record position, advancement, and transfer, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.24. READONLY Specifier

The READONLY specifier indicates only READ statements can refer to this connection. It takes the following form:

```
READONLY
```

READONLY is similar to specifying ACTION='READ', but READONLY prevents deletion of the file if it is closed with STATUS='DELETE' in effect.

Default **file access** privileges are READWRITE, which can cause run-time I/O errors if the file protection does not permit write access.

The READONLY specifier has no effect on the protection specified for a file. Its main purpose is to allow a file to be read simultaneously by two or more programs. For example, use READONLY if you wish to open a file so you can read it, but you also want others to be able to read the same file while you have it open.

For More Information:

On file sharing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.25. RECL Specifier

The RECL specifier indicates the length of each record in a file connected for direct or keyed access, or the maximum length of a record in a file connected for sequential access.

The RECL specifier takes the following form:

```
RECL = rl
```

rl

Is a positive numeric expression indicating the length of records in the file. If necessary, the value is converted to integer data type before use.

If the file is connected for formatted data transfer, the value must be expressed in bytes (characters). Otherwise, the value is expressed in 4-byte units (longwords). If the file is connected for unformatted data transfer, the value can be expressed in bytes if compiler option /ASSUME=BYTERECL is specified.

The *rl* value is the length for record data only. It does not include space for control information, such as two segment control bytes (if present) or the bytes that RMS requires for maintaining record length and deleted record control information.

The length specified is interpreted depending on the type of records in the connected file, as follows:

- For segmented records, RECL indicates the maximum length for any segment (not including the two segment control bytes).
- For fixed-length records, RECL indicates the size of each record.
- For variable-length or stream records, RECL specifies the size of the buffer that will be allocated to hold records read or written. Specifying RECL for stream records (STREAM, STREAM_CR or STREAM_LF) is required if the longest record length in the file exceeds the default RECL value.

Errors occur under the following conditions:

- If your program attempts to write to an existing file a record that is longer than the logical record length
- If you are opening an existing file that contains fixed-length records or has relative organization and you specify a value for RECL that is different from the actual length of the records in the file

Table 12.2 lists the maximum values that can be specified for *rl* for disk files that use the fixed-length record format:

Table 12.2. Maximum Record Lengths (RECL)

File Organization	Record I/O Statement Format	
	Formatted (bytes)	**Unformatted (longwords)**
Sequential	32767	8191
Relative	32255	8063
Indexed	32224	8056

For other record formats and device types, the record size limit may be less, as described in the *OpenVMS Record Management Services Reference Manual*.

You must specify RECL when opening new files (STATUS='NEW','UNKNOWN', or'SCRATCH') and one or more of the following conditions exists:

- The file is connected for direct access (ACCESS='DIRECT').
- The record format is fixed length (RECORDTYPE='FIXED').
- The file organization is relative or indexed (ORGANIZATION='RELATIVE' or 'INDEXED').

The default value depends on the setting of the RECORDTYPE specifier, as shown in Table 12.3.

Table 12.3. Default Record Lengths (RECL)

RECORDTYPE	RECL value
'FIXED'	None; value must be explicitly specified
All other types	133 bytes (for formatted records) 511 longwords (for unformatted records)

12.6.26. RECORDSIZE Specifier

RECORDSIZE is a nonstandard synonym for RECL (see Section 12.6.25).

12.6.27. RECORDTYPE Specifier

The RECORDTYPE specifier indicates the type of records in a file. It takes the following form:

```
RECORDTYPE = typ
```

typ

Is a scalar default character expression that evaluates to one of the following values:

'FIXED'	Indicates fixed-length records.
'VARIABLE'	Indicates variable-length records.
'SEGMENTED'	Indicates segmented records.
'STREAM'	Indicates stream-type variable length records.
'STREAM_CR'	Indicates stream-type variable length records, terminated with a carriage-return.
'STREAM_LF'	Indicates stream-type variable length records, terminated with a line feed.

When you open a file, default record types are as follows:

'FIXED'	For relative or indexed files
'FIXED'	For direct access sequential files
'VARIABLE'	For formatted sequential access files
'SEGMENTED'	For unformatted sequential access files

A **segmented record** is a logical record consisting of one or more variable-length records (segments). The logical record can span several physical records. Only unformatted sequential-access files with sequential organization can have segmented records;'SEGMENTED' must not be specified for any other file type.

Files containing segmented records can be accessed only by unformatted sequential data transfer statements.

Normally, if you do not use the RECORDTYPE specifier when you are accessing an existing file, the record type of the file is used. However, if the file is an unformatted sequential-access file with sequential organization and variable-length records, the default record type is 'SEGMENTED'.

If you use the RECORDTYPE specifier when you are accessing an existing file, the type that you specify must match the type of the existing file.

If an output statement does not specify a full record for a file containing fixed-length records, the following occurs:

- In formatted files, the record is filled with blanks
- In unformatted files, the record is filled with zeros

For More Information:

On record types and file organization, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.28. SHARED Specifier

The SHARED specifier indicates that the file is connected for shared access by more than one program executing simultaneously. It takes the following form:

```
SHARED
```

For More Information:

On file sharing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.6.29. STATUS Specifier

The STATUS specifier indicates the status of a file when it is opened. It takes the following form:

```
STATUS = sta
```

sta

Is a scalar default character expression that evaluates to one of the following values:

'OLD'	Indicates an existing file.
'NEW'	Indicates a new file; if the file already exists, an error occurs. Once the file is created, its status changes to'OLD'.
'SCRATCH'	Indicates a new file that is unnamed (called a scratch file). When the file is closed or the program terminates, the scratch file is deleted.
'REPLACE'	Indicates the file replaces another. If the file to be replaced exists, it is deleted and a new file is created with the same name. If the file to be replaced does not exist, a new file is created and its status changes to'OLD'.
'UNKNOWN'	Indicates the file may or may not exist. If the file does not exist, a new file is created (using the next highest available version number) and its status changes to'OLD'.

The default is 'UNKNOWN'. However, if you implicitly open a file using WRITE or you specify compiler option /NOF77 (or OPTIONS /NOF77), the default value is 'NEW'. If you implicitly open a file using READ, the default value is 'OLD'.

Scratch files (STATUS='SCRATCH') are created on your default disk (SYS$DISK) and are not placed in a directory or given a name that is externally visible. To indicate a different device, use the FILE specifier.

Note

The STATUS specifier can also appear in CLOSE statements to indicate the file's status after it is closed. However, in CLOSE statements the STATUS values are the same as those listed for the DISPOSE specifier (see Section 12.6.12).

12.6.30. TYPE Specifier

TYPE is a nonstandard synonym for STATUS (see Section 12.6.29).

12.6.31. USEROPEN Specifier

The USEROPEN specifier indicates a user-written external function that controls the opening of the file. It takes the following form:

```
USEROPEN = function-name
```

`function-name`

Is the name of the user-written function to receive control.

The function must be declared in a previous EXTERNAL statement; if it is typed, it must be of type INTEGER(4) (INTEGER*4).

The USEROPEN specifier lets experienced users use additional features of the operating system that are not normally available in Fortran.

For More Information:

On user-supplied functions to use with USEROPEN, including examples, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.7. REWIND Statement

The REWIND statement positions a sequential or direct access file at the beginning of the file (the initial point). It takes one of the following forms:

```
REWIND ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var])
REWIND io-unit
```

`io-unit`

Is an external unit specifier.

`label`

Is the label of the branch target statement that receives control if an error occurs.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The unit number must refer to a file on disk or magnetic tape, and the file must be open for sequential, direct, or append access.

If a REWIND is done on a direct access file, the NEXTREC specifier is assigned a value of 1.

A REWIND statement must not be specified for a file that is open for or keyed access.

If a file is already positioned at the initial point, a REWIND statement has no effect.

If a REWIND statement is specified for a unit that is not open, it has no effect.

Examples

The following statement repositions the file connected to I/O unit 3 to the beginning of the file:

```
REWIND 3
```

Consider the following statement:

```
REWIND (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 at the beginning of the file. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On record position, advancement, and transfer, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

12.8. UNLOCK Statement

The UNLOCK statement frees a record in an indexed, relative, or sequential file that was locked by a previous READ statement.

The UNLOCK statement takes one of the following forms:

```
UNLOCK ([UNIT=]io-unit [,ERR=label] [,IOSTAT=i-var])
UNLOCK io-unit
```

`io-unit`

Is an external unit specifier.

`label`

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

If no record is locked, the UNLOCK statement has no effect.

Examples

The following statement frees any record previously read and locked in the file connected to I/O unit 4:

```
UNLOCK 4
```

Consider the following statement:

```
UNLOCK (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement frees any record previously read and locked in the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

For More Information:

- On the UNIT control specifier, see Section 10.2.1.1.
- On the ERR control specifier, see Section 10.2.1.8.
- On the IOSTAT control specifier, see Section 10.2.1.7.
- On shared files and locked records, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Chapter 13. Compilation Control Statements

In addition to specifying options on the compiler command line, you can specify the following statements in a program unit to influence compilation:

- DICTIONARY statement (Section 13.1)

  Extracts records from the Common Data Dictionary (CDD) and converts them into VSI Fortran STRUCTURE declarations.

- INCLUDE statement (Section 13.2)

  Incorporates external source code into programs.

- OPTIONS statement (Section 13.3)

  Sets options usually specified in the compiler command line. OPTIONS statement settings override command line options.

13.1. DICTIONARY Statement

The DICTIONARY statement incorporates common data dictionary (CDD) data definitions into the current VSI Fortran source program during compilation. The statement can occur any place in a Fortran source program where a STRUCTURE statement can occur.

The DICTIONARY statement takes the following form:

```
DICTIONARY 'cdd-path [/[NO]LIST]'
```

cdd-path

Is interpreted as the full or relative pathname of a CDD object.

/[NO]LIST

Controls whether the source code representation of the resulting structure declaration is listed in a compilation source listing. The default is /NOLIST. /LIST and /NOLIST must be spelled completely.

Rules and Behavior

There are two types of CDD pathnames: full and relative. Their form must conform to the rules for forming CDD pathnames.

A full CDD pathname begins with CDD$TOP and specifies the given names of all its descendants; it is a complete path to the record definition. Multiple descendant names are separated by periods.

A relative CDD pathname begins with any generation name other than CDD$TOP and specifies the given names of the descendants after that point. A relative path comes into existence when a default directory is established with a logical name.

Examples

In the following example, the logical name definition specifies the beginning of the CDD pathname. So, a relative pathname specifies the remainder of the path to the record definition:

```
$ DEFINE CDD$DEFAULT CDD$TOP.FOR
```

The following examples show how a CDD pathname beginning with CDD$TOP overrides the default CDD pathname. Consider a record with the pathname CDD$TOP.SALES.JONES.SALARY. If you define CDD$DEFAULT to be CDD$TOP.SALES.JONES, you can then specify a relative pathname; for example:

```
DICTIONARY 'SALARY'
```

You can also specify this as a full pathname, for example:

```
DICTIONARY 'CDD$TOP.SALES.JONES.SALARY'
```

For More Information:

On CDD pathnames, see *Using CDD/Repository on VMS Systems*.

13.2. INCLUDE Statement

The INCLUDE statement directs the compiler to stop reading statements from the current file and read statements in an included file or text module.

The INCLUDE statement takes one of the following forms:

```
INCLUDE 'file-name [/[NO]LIST]'
INCLUDE '[text-lib] (module-name) [/[NO]LIST]'
```

file-name

Is a character string specifying the name of the file to be included; it must not be a named constant.

The form of the file name must be acceptable to the operating system, as described in your system documentation.

/[NO]LIST

Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the “include” nesting depth of the code. The default is /NOLIST. /LIST and /NOLIST must be spelled completely.

text-lib

Is a character string specifying the file name of the text library to be searched.

The form of the file name must be acceptable to the operating system, as described in your system documentation.

module-name

Is a character string specifying the name of the text library module to be included. The name of the text module must be enclosed in parentheses. It can contain any alphanumeric character and the special characters dollar sign ($) and underscore (_).

The length of the file name must be acceptable to the operating system, as described in your system documentation.

Rules and Behavior

An INCLUDE statement can appear anywhere within a scoping unit. The statement can span more than one source line, but no other statement can appear on the same line. The source line cannot be labeled.

An included file or text module cannot begin with a continuation line, and each Fortran statement must be completely contained within a single file.

An included file or text module can contain any source text, but it cannot begin or end with an incomplete Fortran statement.

The included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions shown in Figure 2.1.

Included files or text modules can contain additional INCLUDE statements, but they must not be recursive. INCLUDE statements can be nested until system resources are exhausted.

When the included file or text module completes execution, compilation resumes with the statement following the INCLUDE statement.

When including files that contain datatype declarations, it is recommended that such declarations explicitly specify the kind of the datatype. If an explicit kind is omitted, the declarations will be interpreted according to the command-line options in effect when the file is included, which may result in unintended behavior.

Examples

In Example 13.1, a file named COMMON.FOR (in the current working directory) is included and read as input.

Example 13.1. Including Text from a File

Main Program File

```
PROGRAM
  INCLUDE 'COMMON.FOR'
  REAL, DIMENSION(M) :: Z
  CALL CUBE
  DO I = 1, M
    Z(I) = X(I) + SQRT(Y(I))
    ...
  END DO
END

SUBROUTINE CUBE
  INCLUDE 'COMMON.FOR'
  DO I=1,M
    X(I) = Y(I)**3
  END DO
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

```
  RETURN
END
```

The file COMMON.FOR defines a named constant M, and defines arrays X and Y as part of blank common.

For More Information:

- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On using text libraries, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

13.3. OPTIONS Statement

The OPTIONS statement overrides or confirms the compiler options in effect for a program unit. It takes the following form:

```
OPTIONS option [option...]
```

`option`

Is one of the following:

/CHECK = { ALL | [NO]BOUNDS | [NO]OVERFLOW | [NO]UNDERFLOW | NONE }

/NOCHECK

/CONVERT = { BIG_ENDIAN | CRAY | FDX | FGX | IBM | LITTLE_ENDIAN | NATIVE | VAXD | VAXG }

/[NO]EXTEND_SOURCE

/[NO]F77

/FLOAT = { D_FLOAT | G_FLOAT | IEEE_FLOAT }

/[NO]G_FLOATING

/[NO]I4

/[NO]RECURSIVE

Note that an option must always be preceded by a slash (/).

Some OPTIONS statement options are equivalent to compiler options.

Rules and Behavior

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, MODULE, and BLOCK DATA statements.

OPTIONS statement options override compiler options, but only until the end of the program unit for which they are defined. If you want to override compiler options in another program unit, you must specify the OPTIONS statement before that program unit.

Examples

The following are valid OPTIONS statements:

```
OPTIONS /CHECK=ALL/F77
OPTIONS /I4
```

For More Information:

On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

Chapter 14. Compiler Directives

VSI Fortran provides compiler directives to perform general-purpose tasks during compilation. You do not need to specify a compiler option to enable general directives.

Compiler directives are preceded by a special prefix that identifies them to the compiler.

This chapter describes:

- Syntax rules for general directives (Section 14.1)
- ALIAS (Section 14.2)

  Specifies an alternate external name to be used when referring to external subprograms.
- ATTRIBUTES (Section 14.3)

  Specifies properties for data objects and procedures.
- DECLARE and NODECLARE (Section 14.4)

  Generates or disables warnings for variables that have been used but not declared.
- DEFINE and UNDEFINE (Section 14.5)

  Specifies a symbolic variable whose existence (or value) can be tested during conditional compilation.
- FIXEDFORMLINESIZE (Section 14.6)

  Sets the line length for fixed-form source code.
- FREEFORM and NOFREEFORM (Section 14.7)

  Specifies free-format or fixed-format source code.
- IDENT (Section 14.8)

  Specifies an identifier for an object module.
- IF and IF DEFINED (Section 14.9)

  Specifies a conditional compilation construct.
- INTEGER (Section 14.10)

  Specifies the default integer kind.
- IVDEP (Section 14.11)

  Assists the compiler's dependence analysis.
- MESSAGE (Section 14.12)

  Specifies a character string to be sent to the standard output device during the first compiler pass.
- OBJCOMMENT (Section 14.13)

Specifies a library search path in an object file.

- OPTIONS (Section 14.14)

  Affects data alignment and warnings about data alignment.

- PACK (Section 14.15)

  Specifies the memory starting addresses of derived-type items.

- PSECT (Section 14.16)

  Modifies certain characteristics of a common block.

- REAL (Section 14.17)

  Specifies the default real kind.

- STRICT and NOSTRICT (Section 14.18)

  Disables or enables language features not found in the language standard specified on the command line (Fortran 95 or Fortran 90).

- TITLE and SUBTITLE (Section 14.19)

  Specifies a title or subtitle for a listing header.

- UNROLL (Section 14.20)

  Tells the compiler's optimizer how many times to unroll a DO loop.

14.1. Syntax Rules for General Directives

The following general syntax rules apply to all general compiler directives. You must follow these rules precisely to compile your program properly and obtain meaningful results.

A general directive prefix (tag) takes the following form:

```
cDEC$
```

c

Is one of the following: C (or c), !, or *.

The following are source form rules for directive prefixes:

- Prefixes beginning with C (or c) and * are only allowed in fixed and tab source forms.

  In these source forms, the prefix must appear in columns 1 through 5; column 6 must be a blank or tab. From column 7 on, blanks are insignificant, so the directive can be positioned anywhere on the line after column 6.

- Prefixes beginning with ! are allowed in all source forms.

  The prefix can appear in any valid column, but it cannot be preceded by any nonblank characters on the same line. It can only be preceded by whitespace.

A general directive ends in column 72 (or column 132, if a compiler option is specified).

General directives cannot be continued.

A comment can follow a directive on the same line.

Additional Fortran statements (or directives) cannot appear on the same line as the general directive.

General directives cannot appear within a continued Fortran statement.

If a blank common is used in a general compiler directive, it must be specified as two slashes (/ /).

14.2. ALIAS Directive

The ALIAS directive lets you specify an alternate external name to be used when referring to external subprograms. This can be useful when compiling applications written for other platforms that have different naming conventions.

The ALIAS directive takes the following form:

```
cDEC$ ALIAS internal-name, external-name
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

internal-name

Is the name of the subprogram as used in the current program unit.

external-name

Is a name, or a character constant delimited by apostrophes or quotation marks.

If a name is specified, the name (in uppercase) is used as the external name for the specified *internal-name*. If a character constant is specified, it is used as is; the string is not changed to uppercase, nor are blanks removed.

The ALIAS directive affects only the external name used for references to the specified *internal-name*.

Names that are not acceptable to the linker will cause link-time errors.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the linker, see the *VSI OpenVMS Linker Utility Manual*.

14.3. ATTRIBUTES Directive

The ATTRIBUTES directive lets you specify properties for data objects and procedures. It takes the following form:

```
cDEC$ ATTRIBUTES att [,att]... :: object [,object]...
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

att

Is one of the following:

ADDRESS64	DESCRIPTOR32	REFERENCE
ALIAS	DESCRIPTOR64	REFERENCE32
ALLOW_NULL		REFERENCE64
		STDCALL
C	EXTERN	VALUE
DECORATE	IGNORE_LOC	VARYING
DEFAULT	NO_ARG_CHECK	
DESCRIPTOR	NOMIXED_STR_LEN_ARG	

object

Is the name of a data object or procedure.

The following table shows which properties can be used with various objects:

Property	**Variable and Array Declarations**	**Common Block Names**[1]	**Subprogram Specification and EXTERNAL Statements**
ADDRESS64	Yes	Yes	No
ALIAS	No	Yes	Yes
ALLOW_NULL	Yes	No	No
C	No	Yes	Yes
DECORATE	No	No	Yes
DEFAULT	No	Yes	Yes
DESCRIPTOR	Yes[2]	No	No
DESCRIPTOR32	Yes[2]	No	No
DESCRIPTOR64	Yes[2]	No	No
EXTERN	Yes	No	No
IGNORE_LOC	Yes	No	No
NO_ARG_CHECK	Yes	No	Yes[3]
NOMIXED_STR_LEN_ARG	No	No	Yes
REFERENCE	Yes	No	Yes
REFERENCE32	Yes	No	No
REFERENCE64	Yes	No	No
STDCALL	No	Yes	Yes
VALUE	Yes	No	No
VARYING	No	No	Yes

[1] A common block name is specified as [/]common-block-name[/].

[2]This property can only be applied to INTERFACE blocks.
[3]This property cannot be applied to EXTERNAL statements.

These properties can be used in function and subroutine definitions, in type declarations, and with the INTERFACE and ENTRY statements.

Properties applied to entities available through use or host association are in effect during the association. For example, consider the following:

```
MODULE MOD1
  INTERFACE
    SUBROUTINE SUB1
    !DEC$ ATTRIBUTES C, ALIAS:'othername' :: NEW_SUB
    END SUBROUTINE
  END INTERFACE
  CONTAINS
    SUBROUTINE SUB2
    CALL NEW_SUB
    END SUBROUTINE
END MODULE
```

In this case, the call to NEW_SUB within SUB2 uses the C and ALIAS properties specified in the interface block.

The properties are described as follows:

- ADDRESS64

  Specifies that the object has a 64-bit address. This property can be specified for any variable or dummy argument, including ALLOCATABLE and deferred-shape arrays. However, variables with this property cannot be data-initialized.

  It can also be specified for COMMON blocks or for variables in a COMMON block. If specified for a COMMON block variable, the COMMON block implicitly has the ADDRESS64 property.

  ADDRESS64 is not compatible with the AUTOMATIC attribute.

- ALIAS

  Specifies an alternate external name to be used when referring to external subprograms. Its form is:

  ```
  ALIAS:external-name
  ```

  external-name

  Is a character constant delimited by apostrophes or quotation marks. The character constant is used as is; the string is not changed to uppercase, nor are blanks removed.

  The ALIAS property overrides the C (and STDCALL) property. If both C and ALIAS are specified for a subprogram, the subprogram is given the C calling convention, but not the C naming convention. It instead receives the name given for ALIAS, with no modifications.

  ALIAS cannot be used with internal procedures, and it cannot be applied to dummy arguments.

  cDEC$ ATTRIBUTES ALIAS has the same effect as the cDEC$ ALIAS directive (see Section 14.2).

- ALLOW_NULL

Enables a corresponding dummy argument to pass a NULL pointer (defined by a zero or the NULL intrinsic function) by value for the argument.

ALLOW_NULL is only valid if the REFERENCE property is also specified; otherwise, it has no effect.

- C and STDCALL

  Specify how data is to be passed when you use routines written in C or assembler with FORTRAN or Fortran 95/90 routines.

  C and STDCALL are synonyms.

  When applied to a subprogram, these properties define the subprogram as having a specific set of calling conventions.

  The following table summarizes the differences between the calling conventions:

Convention	**C**[1]	**STDCALL**[1]	**Default**[2]
Arguments passed by value	Yes	Yes	No

[1]C and STDCALL are synonyms.

[2]Fortran 95/90 calling convention.

  If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran 95/90 conventions pass arguments by reference.

  Character arguments are passed as follows:

  - By default, hidden lengths are put at the end of the argument list.
  - If C or STDCALL (only) is specified, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
  - If C or STDCALL is specified, and REFERENCE is specified for the argument, the string is passed but the length is not passed.
  - If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is *not* specified for the argument, if any), the string is passed but the length is not passed.

  For details, see information on mixed-language programming in the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/]. See also the description of REFERENCE in this list.

- DECORATE

  Specifies that the external name used in cDEC$ ALIAS or cDEC$ ATTRIBUTES ALIAS should have the prefix and postfix decorations performed on it that are associated with the calling mechanism that is in effect. These are the same decorations performed on the procedure name when ALIAS is not specified.

  The case of the external name is not modified.

  If ALIAS is not specified, this property has no effect.

See also the summary of prefix and postfix decorations in the above description of ATTRIBUTES options C and STDCALL.

- DEFAULT

  Overrides certain compiler options that can affect external routine and COMMON block declarations.

  It specifies that the compiler should ignore compiler options that change the default conventions for external symbol naming and argument passing for routines and COMMON blocks.

  This option can be combined with other cDEC$ ATTRIBUTES options, such as STDCALL, C, REFERENCE, ALIAS, etc. to specify attributes different from the compiler defaults.

  This option is useful when declaring INTERFACE blocks for external routines, since it prevents compiler options from changing calling or naming conventions.

- DESCRIPTOR

  Specifies that the argument is passed by VMS descriptor. This property can be specified only for dummy arguments in an INTERFACE block (*not* for a routine name).

- DESCRIPTOR32

  Specifies that the argument is passed as a 32-bit descriptor.

- DESCRIPTOR64

  Specifies that the argument is passed as a 64-bit descriptor.

- EXTERN

  Specifies that a variable is allocated in another source file. EXTERN can be used in global variable declarations, but it must not be applied to dummy arguments.

  EXTERN must be used when accessing variables declared in other languages.

- IGNORE_LOC

  Enables %LOC to be stripped from an argument.

  IGNORE_LOC is only valid if the REFERENCE property is also specified; otherwise, it has no effect.

- NO_ARG_CHECK

  Specifies that type and shape matching rules related to explicit interfaces are to be ignored. This permits the construction of an INTERFACE block for an external procedure or a module procedure that accepts an argument of any type or shape; for example, a memory copying routine.

  NO_ARG_CHECK can appear only in an INTERFACE block for a non-generic procedure or in a module procedure. It can be applied to an individual dummy argument name or to the routine name, in which case the property is applied to all dummy arguments in that interface.

  NO_ARG_CHECK cannot be used for procedures with the PURE or ELEMENTAL prefix. If an argument has an INTENT or OPTIONAL attribute, any NO_ARG_CHECK specification is ignored.

- NOMIXED_STR_LEN_ARG

  Specifies that hidden lengths be placed in sequential order at the end of the argument list.

- REFERENCE and VALUE

  Specify how a dummy argument is to be passed.

  REFERENCE specifies a dummy argument's memory location is to be passed instead of the argument's value.

  VALUE specifies a dummy argument's value is to be passed instead of the argument's memory location.

  When a dummy argument has the VALUE property, the actual argument passed to it can be of a different type. If necessary, type conversion is performed before the subprogram is called.

  When a complex (KIND=4 or KIND=8) argument is passed by value, *two* floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value.

  Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value.

  If REFERENCE (only) is specified for a character argument, the string is passed but the length is not passed.

  If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine.

  If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has *not* been specified for the argument, the string is passed with the length.

  VALUE is the default if the C or STDCALL property is specified in the subprogram definition.

  For more details, see information on mixed-language programming in the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

- REFERENCE32

  Specifies that the argument is accepted only by 32-bit address.

- REFERENCE64

  Specifies that the argument is accepted only by 64-bit address.

- VARYING

  Allows a variable number of calling arguments. If VARYING is specified, the C property must also be specified.

Either the first argument must be a number indicating how many arguments to process, or the last argument must be a special marker (such as –1) indicating it is the final argument. The sequence of the arguments, and types and kinds must be compatible with the called procedure.

Options C, STDCALL, REFERENCE, VALUE, and VARYING affect the calling conventions of routines:

- You can specify C, STDCALL, REFERENCE, and VARYING for an entire routine.
- You can specify VALUE and REFERENCE for individual arguments.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On using the cDEC$ ATTRIBUTES directive, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

14.4. DECLARE or NODECLARE Directives

The DECLARE directive generates warnings for variables that have been used but have not been declared (like the IMPLICIT NONE statement). The NODECLARE directive (the default) disables these warnings.

The DECLARE and NODECLARE directives take the following forms:

```
cDEC$ DECLARE
cDEC$ NODECLARE
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

The DECLARE directive is primarily a debugging tool that locates variables that have not been properly initialized, or that have been defined but never used.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the IMPLICIT NONE statement, see Section 5.9.

14.5. DEFINE and UNDEFINE Directives

The DEFINE directive creates a symbolic variable whose existence or value can be tested during conditional compilation. The UNDEFINE directive removes a defined symbol.

The DEFINE and UNDEFINE directives take the following forms:

```
cDEC$ DEFINE name [=val]
cDEC$ UNDEFINE name
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

name

Is the name of the variable.

val

Is an INTEGER(4) value assigned to *name*.

Rules and Behavior

DEFINE and UNDEFINE create and remove variables for use with the IF (or IF DEFINED) directive. Symbols defined with the DEFINE directive are local to the directive. They cannot be declared in the Fortran program.

Because Fortran programs cannot access the named variables, the names can duplicate Fortran keywords, intrinsic functions, or user-defined names without conflict.

To test whether a symbol has been defined, use the IF DEFINED (name) directive You can assign an integer value to a defined symbol. To test the assigned value of *name*, use the IF directive. IF test expressions can contain most logical and arithmetic operators.

Attempting to undefine a symbol that has not been defined produces a compiler warning.

The DEFINE and UNDEFINE directives can appear anywhere in a program, enabling and disabling symbol definitions.

Examples

Consider the following:

```
!DEC$ DEFINE  testflag
!DEC$ IF DEFINED (testflag)
    WRITE (*,*) 'Compiling first line'
!DEC$ ELSE
    WRITE (*,*) 'Compiling second line'
!DEC$ ENDIF
!DEC$ UNDEFINE  testflag
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the IF and IF DEFINED directives, see Section 14.9.

14.6. FIXEDFORMLINESIZE Directive

The FIXEDFORMLINESIZE directive sets the line length for fixed-form source code. The directive takes the following form:

```
cDEC$ FIXEDFORMLINESIZE:{72 | 80 | 132}
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

You can set FIXEDFORMLINESIZE to 72 (the default), 80, or 132 characters. The FIXEDFORMLINESIZE setting remains in effect until the end of the file, or until it is reset.

The FIXEDFORMLINESIZE directive sets the source-code line length in include files, but not in USE modules, which are compiled separately. If an include file resets the line length, the change does not affect the host file.

This directive has no effect on free-form source code.

Examples

Consider the following:

```
CDEC$ NOFREEFORM
CDEC$ FIXEDFORMLINESIZE:132
WRITE(*,*) 'Sentence that goes beyond the 72nd column without
 continuation.'
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On fixed-format source code, see Section 2.3.2.

14.7. FREEFORM and NOFREEFORM Directives

The FREEFORM directive specifies that source code is in free-form format. The NOFREEFORM directive specifies that source code is in fixed-form format.

These directives take the following forms:

```
cDEC$ FREEFORM
cDEC$ NOFREEFORM
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

When the FREEFORM or NOFREEFORM directives are used, they remain in effect for the remainder of the file, or until the opposite directive is used. When in effect, they apply to include files, but do not affect USE modules, which are compiled separately.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On free-form and fixed-form source code, see Section 2.3.

14.8. IDENT Directive

The IDENT directive specifies a string that identifies an object module. The compiler places the string in the identification field of an object module when it generates the module for each source program unit. The IDENT directive takes the following form:

```
cDEC$ IDENT string
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

string

Is a character constant containing up to 31 printable characters.

Only the first IDENT directive is effective; the compiler ignores any additional IDENT directives in a program unit or module.

For More Information:

On syntax rules for all general directives, see Section 14.1.

14.9. IF and IF DEFINED Directives

The IF and IF DEFINED directives specify a conditional compilation construct. IF tests whether a logical expression is .TRUE. or .FALSE.. IF DEFINED tests whether a symbol has been defined.

The directive-initiated construct takes the following form:

```
cDEC$ IF (expr) [or cDEC$ IF DEFINED (name)]
   block
[cDEC$ ELSE IF (expr)
   block]...
[cDEC$ ELSE
   block]
cDEC$ ENDIF
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

expr

Is a logical expression that evaluates to .TRUE. or .FALSE..

name

Is the name of a symbol to be tested for definition.

block

Are executable statements that are compiled (or not) depending on the value of logical expressions in the IF directive construct.

Rules and Behavior

The IF and IF DEFINED directive constructs end with an ENDIF directive and can contain one or more ELSEIF directives and at most one ELSE directive. If the logical condition within a directive evaluates to .TRUE. at compilation, and all preceding conditions in the IF construct evaluate to .FALSE., then the statements contained in the directive block are compiled.

A *name* can be defined with a DEFINE directive, and can optionally be assigned an integer value. If the symbol has been defined, with or without being assigned a value, IF DEFINED (name) evaluates to .TRUE.; otherwise, it evaluates to .FALSE..

If the logical condition in the IF or IF DEFINED directive is .TRUE., statements within the IF or IF DEFINED block are compiled. If the condition is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If the logical expression in an ELSEIF directive is .TRUE., statements within the ELSEIF block are compiled. If the expression is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If control reaches an ELSE directive because all previous logical conditions in the IF construct evaluated to .FALSE., the statements in an ELSE block are compiled unconditionally.

You can use any Fortran logical or relational operator or symbol in the logical expression of the directive, including: .LT., <, .GT., >, .EQ., ==, .LE., <=, .GE., >=, .NE., /=, .EQV., .NEQV., .NOT., .AND., .OR., and .XOR.. The logical expression can be as complex as you like, but the whole directive must fit on one line.

Examples

Consider the following:

```
! When the following code is compiled and run,
!  the output depends on whether one of the expressions
!  tests .TRUE., or all test .FALSE.

!DEC$ DEFINE flag=3
!DEC$ IF (flag .LT. 2)
   WRITE (*,*) "This is compiled if flag less than 2."
!DEC$ ELSEIF (flag >= 8)
   WRITE (*,*) "Or this compiled if flag greater than &
                or equal to 8."
!DEC$ ELSE
   WRITE (*,*) "Or this compiled if all preceding &
                conditions .FALSE."
!DEC$ ENDIF
END
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the DEFINE and UNDEFINE directives, see Section 14.5.

14.10. INTEGER Directive

The INTEGER directive specifies the default integer kind. This directive takes the following form:

```
cDEC$ INTEGER:{1 | 2 | 4 | 8}
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

Rules and Behavior

The INTEGER directive specifies a size of 1 (KIND=1), 2 (KIND=2), 4 (KIND=4), or 8 (KIND=8) bytes for default integer numbers.

When the INTEGER directive is effect, all default integer variables are of the kind specified in the directive. Only numbers specified or implied as INTEGER without KIND are affected.

The INTEGER directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. The directive cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

The default logical kind is the same as the default integer kind. So, when you change the default integer kind you also change the default logical kind.

Examples

Consider the following:

```
INTEGER i               ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )
WRITE(*,*) KIND(i)      ! still a 4-byte integer
                        !   not affected by setting in subroutine
END
SUBROUTINE INTEGER2( )
   !DEC$ INTEGER:2
   INTEGER j            ! a 2-byte integer
   WRITE(*,*) KIND(j)
END SUBROUTINE
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the INTEGER data type, see Section 3.2.1.
- On the REAL directive, see Section 14.17.

14.11. IVDEP Directive

The IVDEP directive assists the compiler's dependence analysis. It can only be applied to iterative DO loops. This directive can also be specified as INIT_DEP_FWD (INITialize DEPendences ForWarD).

The IVDEP directive takes the following form:

```
cDEC$ IVDEP
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

Rules and Behavior

The IVDEP directive is an assertion to the compiler's optimizer about the order of memory references inside a DO loop.

The IVDEP directive tells the compiler to begin dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

The IVDEP directive must precede the DO statement for each DO loop it affects. No source code lines, other than the following, can be placed between the IVDEP directive statement and the DO statement:

- An UNROLL directive
- Placeholder lines
- Comment lines
- Blank lines

The IVDEP directive is applied to a DO loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is “earlier” than the left-hand side.

The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Examples

In the following example, the IVDEP directive provides more information about the dependences within the loop, which may enable loop transformations to occur:

```
!DEC$ IVDEP
      DO I=1, N
         A(INDARR(I)) = A(INDARR(I)) + B(I)
      END DO
```

In this case, the scalar execution order follows:

1. Retrieve INDARR(I).
2. Use the result from step 1 to retrieve A(INDARR(I)).
3. Retrieve B(I).

4. Add the results from steps 2 and 3.

5. Store the results from step 4 into the location indicated by A(INDARR(I)) from step 1.

IVDEP directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.

For More Information:

On syntax rules for all general directives, see Section 14.1.

14.12. MESSAGE Directive

The MESSAGE directive specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.

This directive takes the following form:

```
cDEC$ MESSAGE:string
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

string

Is a character constant specifying a message.

Examples

Consider the following:

```
!DEC$ MESSAGE:'Compiling Sound Speed Equations'
```

For More Information:

On syntax rules for all general directives, see Section 14.1.

14.13. OBJCOMMENT Directive

The OBJCOMMENT directive specifies a library search path in an object file. This directive takes the following form:

```
cDEC$ OBJCOMMENT LIB:library
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

library

Is a character constant specifying the name and, if necessary, the path of the library that the linker is to search.

Rules and Behavior

The linker searches for the library named by the OBJCOMMENT directive as if you named it on the command line, that is, before default library searches. You can place multiple library search directives in the same source file. Each search directive appears in the object file in the order it is encountered in the source file.

If the OBJCOMMENT directive appears in the scope of a module, any program unit that uses the module also contains the directive, just as if the OBJCOMMENT directive appeared in the source file using the module.

If you want to have the OBJCOMMENT directive in a module, but do not want it in the program units that use the module, place the directive outside the module that is used.

Examples

Consider the following:

```
! MOD1.F90
MODULE a
   !DEC$ OBJCOMMENT LIB: "opengl32.lib"
END MODULE a

! MOD2.F90
!DEC$ OBJCOMMENT LIB: "graftools.lib"
MODULE b
   !
END MODULE b

! USER.F90
PROGRAM go
   USE a       ! library search contained in MODULE a
               !   included here
   USE b       ! library search not included
END
```

For More Information:

On syntax rules for all general directives, see Section 14.1.

14.14. OPTIONS Directive

The OPTIONS directive affects data alignment and warnings about data alignment. It takes the following form:

```
cDEC$ OPTIONS option [option]
. . .
cDEC$ END OPTIONS
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

option

Is one (or both) of the following:

- /WARN=[NO]ALIGNMENT

  Controls whether warnings are issued by the compiler for data that is not naturally aligned. By default, you receive compiler messages when misaligned data is encountered (/WARN=ALIGNMENT).

- /[NO]ALIGN[=p]

  Controls alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

  #### p

  Is a specifier with one of the following forms:

  { [class =] rule | (class = rule,...) | ALL | NONE }

  #### class

  Is one of the following keywords:

  - COMMONS: For common blocks
  - RECORDS: For records
  - STRUCTURES: A synonym for RECORDS

  #### rule

  Is one of the following keywords:

  - PACKED

    Packs fields in records or data items in common blocks on arbitrary byte boundaries.

  - NATURAL

    Naturally aligns fields in records and data items in common blocks on up to 64-bit boundaries (inconsistent with the Fortran 95/90 standard).

    This keyword causes the compiler to naturally align all data in a common block, including INTEGER(8), REAL(8), and all COMPLEX data.

  - STANDARD

    Naturally aligns data items in common blocks on up to 32-bit boundaries (consistent with the Fortran 95/90 standard).

    This keyword only applies to common blocks; so, you can specify /ALIGN=COMMONS=STANDARD, but you cannot specify /ALIGN=STANDARD.

  #### ALL

Is the same as specifying /ALIGN, /ALIGN=NATURAL, and /ALIGN=(RECORDS=NATURAL,COMMONS=NATURAL).

NONE

Is the same as specifying /NOALIGN, /ALIGN=PACKED, and /ALIGN=(RECORDS=PACKED,COMMONS=PACKED).

Rules and Behavior

The OPTIONS (and accompanying END OPTIONS) directives must come after OPTIONS, SUBROUTINE, FUNCTION, and BLOCK DATA statements (if any) in the program unit, and before the executable part of the program unit.

The OPTIONS directive supersedes the compiler option that sets alignment and the compiler option that sets warnings about alignment.

For performance reasons, VSI Fortran aligns local data items on natural boundaries. However, EQUIVALENCE, COMMON, RECORD, and STRUCTURE data declaration statements can force misaligned data. If /WARN=NOALIGNMENT is specified, warnings will not be issued if misaligned data is encountered.

Note

Misaligned data significantly increases the time it takes to execute a program. As the number of misaligned fields encountered increases, so does the time needed to complete program execution. Specifying cDEC$ OPTIONS/ALIGN (or the compiler option that sets alignment) minimizes misaligned data.

If you want aligned data in common blocks, do one of the following:

- Specify /ALIGN=COMMONS=STANDARD for data items up to 32 bits in length.
- Specify /ALIGN=COMMONS=NATURAL for data items up to 64 bits in length.
- Place source data declarations within the common block in descending size order, so that each data item is naturally aligned.

If you want packed, unaligned data in a record structure, do one of the following:

- Specify /ALIGN=RECORDS=PACKED.
- Place source data declarations in the record structure so that the data is naturally aligned.

If you want to pad the size of a common block, use the /ALIGNMENT=COMMON=PAD_ALIGN_SIZE qualifier. It ensures that the padding appended to the common blocks makes the program section size allocation as large as the alignment size.

Note that using the PAD_ALIGN_SIZE keyword results in a program section allocation incompatible with sharing across multiple images if those images are not Fortran images which have also been compiled with this keyword.

An OPTIONS directive must be accompanied by an END OPTIONS directive; the directives can be nested up to 100 levels. For example:

```
CDEC$ OPTIONS /ALIGN=PACKED          ! Start of Group A
   declarations
CDEC$ OPTIONS /ALIGN=RECO=NATU         ! Start of nested Group B
   more declarations
CDEC$ END OPTIONS                      ! End of Group B
   still more declarations
CDEC$ END OPTIONS                    ! End of Group A
```

The CDEC$ OPTIONS specification for Group B only applies to RECORDS; common blocks within Group B will be PACKED. This is because COMMONS retains the previous setting (in this case, from the Group A specification).

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On alignment and data sizes, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

14.15. PACK Directive

The PACK directive specifies the memory starting addresses of derived-type or record structure items. This directive takes the following form:

```
cDEC$ PACK:[{1 | 2 | 4}]
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

Rules and Behavior

Items of derived types and record structures are aligned in memory on the smaller of two sizes: the size of the type of the item, or the current alignment setting. The current alignment setting can be 1, 2, 4, or 8 bytes. The default initial setting is 8 bytes (unless a compiler option specifies otherwise). By reducing the alignment setting, you can pack variables closer together in memory.

The PACK directive lets you control the packing of derived-type or record structure items inside your program by overriding the current memory alignment setting.

For example, if CDEC$ PACK:1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted. If CDEC$ PACK:4 is specified, INTEGER(1), LOGICAL(1), and all character variables begin at the next available byte, whether odd or even. INTEGER(2) and LOGICAL(2) begin on the next even byte; all other variables begin on 4-byte boundaries.

If the PACK directive is specified without a number, packing reverts to the compiler option setting (if any), or the default setting of 8.

The directive can appear anywhere in a program before the derived-type definition or record structure definition. It cannot appear *inside* a derived-type or record structure definition.

Examples

Consider the following:

```
! Use 4-byte packing for this derived type
! Note PACK is used outside of the derived-type definition
!DEC$ PACK:4
TYPE pair
  INTEGER a, b
END TYPE
! revert to default or compiler option
!DEC$ PACK:
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On compiler options that affect packing, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On record structures, see Section B.12.

14.16. PSECT Directive

The PSECT directive modifies several characteristics of a common block. It takes the following form:

```
cDEC$ PSECT /common-name/ a [,a] . . .
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

common-name

Is the name of the common block. The slashes (/) are required.

a

Is one of the following keywords:

- ALIGN= *val* or ALIGN= *keyword*

  Specifies alignment for the common block.

  The *val* is a constant ranging from 0 through 16. The specified number is interpreted as a power of 2. The value of the expression is the alignment in bytes.

  The *keyword* is one of the following:

Keyword	Equivalent to *val*
BYTE	0
WORD	1
LONG	2
QUAD	3
OCTA	4

Keyword	Equivalent to *val*
PAGE[1]	Alpha: 16 I64: 13 x86-64: 13

[1]Range for Alpha is 0 to 16; for IA64 and x86-64, 0 to 13.

- GBL

  Specifies global scope.

- LCL

  Specifies local scope. This keyword is opposite to GBL and cannot appear with it.

- [NO]MULTILANGUAGE

  Controls whether the compiler pads the size of common blocks to ensure compatibility when the common block program section (psect) is shared by code created by other VSI compilers.

  When a program section generated by a Fortran common block is overlaid with a program section consisting of a C structure, linker error messages can occur. This is because the sizes of the program sections are inconsistent; the C structure is padded, but the Fortran common block is not.

  Specifying MULTILANGUAGE ensures that VSI Fortran follows a consistent program section size allocation scheme that works with VSI C program sections shared across multiple images. Program sections shared in a single image do not have a problem.

  You can use a compiler option to specify MULTILANGUAGE for all common blocks in a module.

- [NO]SHR

  Determines whether the contents of a common block can be shared by more than one process.

- [NO]WRT

  Determines whether the contents of a common block can be modified during program execution.

Rules and Behavior

Global or local scope is significant for an image that has more than one cluster. Program sections with the same name that are from different modules in different clusters are placed in separate clusters if local scope is in effect. They are placed in the same cluster if global scope is in effect.

If one program unit changes one or more characteristics of a common block, all other units that reference that common block must also change those characteristics in the same way.

Default characteristics apply if you do not modify them with a PSECT directive. Table 14.1 lists the default characteristics of common blocks and how they can be modified by PSECT.

Table 14.1. Common Block Defaults and PSECT Modification

Default Characteristics	PSECT Modification
Relocatable	None
Overlaid	None

Default Characteristics	PSECT Modification
Global Scope	Global or local scope
Not executable	None
Not multilanguage	Multilanguage or not multilanguage
Writable	Writable or not writable
Readable	None
No protection	None
Octaword alignment[1] (4)	0 through 16[2]
Not shareable	Shareable or not shareable
Position dependent	None

[1]An address that is an integral multiple of 16.

[2]Or keywords BYTE through PAGE.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the default characteristics of common blocks on OpenVMS systems, see the *VSI OpenVMS Linker Utility Manual*.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

14.17. REAL Directive

The REAL directive specifies the default real kind. This directive takes the following form:

```
cDEC$ REAL:{4 | 8 | 16}
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

Rules and Behavior

The REAL directive specifies a size of 4 (KIND=4), 8 (KIND=8), or 16 (KIND=16) bytes for default real numbers.

When the REAL directive is effect, all default real variables are of the kind specified in the directive. Only numbers specified or implied as REAL without KIND are affected.

The REAL directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. The directive cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

Examples

Consider the following:

```
REAL r                ! a 4-byte REAL
WRITE(*,*) KIND(r)
CALL REAL8( )
WRITE(*,*) KIND(r)    ! still a 4-byte REAL
                      !  not affected by setting in subroutine
END
SUBROUTINE REAL8( )
   !DEC$ REAL:8
   REAL s             ! an 8-byte REAL
   WRITE(*,*) KIND(s)
END SUBROUTINE
```

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On the REAL data type, see Section 3.2.2.
- On the INTEGER directive, see Section 14.10.
- On compiler options that can affect REAL types, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

14.18. STRICT and NOSTRICT Directives

The STRICT directive disables language features not found in the language standard specified on the command line (Fortran 95 or Fortran 90). The NOSTRICT directive (the default) enables these language features.

These directives take the following forms:

```
cDEC$ STRICT
cDEC$ NOSTRICT
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 90.

The STRICT and NOSTRICT directives can appear only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. The directives cannot appear between program units, or at the beginning of internal subprograms. They do not affect any modules invoked with the USE statement in the program unit that contains them.

Examples

Consider the following:

```
! NOSTRICT by default
TYPE stuff
   INTEGER(4) k
   INTEGER(4) m
```

```
   CHARACTER(4) name
END TYPE stuff
TYPE (stuff) examp
DOUBLE COMPLEX cd    ! non-standard data type, no error
cd =(3.0D0, 4.0D0)
examp.k = 4          ! non-standard component designation,
                     !   no error
END
SUBROUTINE STRICTDEMO( )
  !DEC$ STRICT
   TYPE stuff
     INTEGER(4) k
     INTEGER(4) m
     CHARACTER(4) name
   END TYPE stuff
   TYPE (stuff) samp
   DOUBLE COMPLEX cd      ! ERROR
   cd =(3.0D0, 4.0D0)
   samp.k = 4             ! ERROR
END SUBROUTINE
```

For More Information:

On syntax rules for all general directives, see Section 14.1.

14.19. TITLE and SUBTITLE Directives

The TITLE directive specifies a string for the title field of a listing header. Similarly, SUBTITLE specifies a string for the subtitle field of a listing header.

These directives take the following forms:

```
cDEC$ TITLE string
cDEC$ SUBTITLE string
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

string

Is a character constant containing up to 31 printable characters.

Rules and Behavior

To enable TITLE and SUBTITLE directives, you must specify the compiler option that produces a source listing file.

When TITLE or SUBTITLE appear on a page of a listing file, the specified string appears in the listing header of the following page.

If two or more of either directive appear on a page, the last directive is the one in effect for the following page.

If neither directive specifies a string, no change occurs in the listing file header.

For More Information:

- On syntax rules for all general directives, see Section 14.1.
- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

14.20. UNROLL Directive

The UNROLL directive tells the compiler's optimizer how many times to unroll a DO loop. This directive can only be applied to iterative DO loops.

The UNROLL directive takes the following form:

```
cDEC$ UNROLL [(n)]
```

c

Is one of the following: C (or c), !, or * (see Section 14.1).

n

Is an integer constant. The range of *n* is 0 through 255.

Rules and Behavior

The UNROLL directive must precede the DO statement for each DO loop it affects. No source code lines, other than the following, can be placed between the UNROLL directive statement and the DO statement:

- An IVDEP directive
- Placeholder lines
- Comment lines
- Blank lines

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

For More Information:

On syntax rules for all general directives, see Section 14.1.

Chapter 15. Scope and Association

15.1. Overview

Program entities are identified by names, labels, input/output unit numbers, operator symbols, or assignment symbols. For example, a variable, a derived type, or a subroutine is identified by its name.

Scope refers to the area in which a name is recognized. A **scoping unit** is the program or part of a program in which a name is defined and known. It can be any of the following:

- Entire executable program
- Single scoping unit
- Single statement (or part of a statement)

The region of the program in which a name is known and accessible is referred to as the scope of that name. These different scopes allow the same name to be used for different things in different regions of the program.

Association is the language concept that allows different names to refer to the same entity in a particular region of a program.

15.2. Scope

Program entities have the following kinds of scope (as shown in Table 15.1):

- Global

  Entities that are accessible throughout an executable program.

  The name of a global entity must be unique. It cannot be used to identify any other global entity in the same executable program.

- Scoping unit (local scope)

  Entities that are declared within a scoping unit.

  These entities are local to that scoping unit. The names of local entities are divided into classes (see Table 15.1).

  A scoping unit is one of the following:

  - Derived-type definition
  - Procedure interface body (excluding any derived-type definitions and interface bodies contained within it)
  - Program unit or subprogram (excluding any derived-type definitions, interface bodies, and subprograms contained within it)

  A scoping unit that immediately surrounds another scoping unit is called the host scoping unit. Named entities within the host scoping unit are accessible to the nested scoping unit by host association. (For information about host association, see Section 15.5.1.2).

Once an entity is declared in a scoping unit, its name can be used throughout that scoping unit. An entity declared in another scoping unit is a different entity even if it has the same name and properties.

Within a scoping unit, a local entity name that is not generic must be unique within its class. However, the name of a local entity in one class can be used to identify a local entity of another class.

Within a scoping unit, a generic name can be the same as any one of the procedure names in the interface block.

A component name has the same scope as the derived type of which it is a component. It can appear only within a component designator of a structure of that type.

For information on interactions between local and global names, see Table 15.1.

- Statement

  Entities that are accessible only within a statement or part of a statement; such entities cannot be referenced in subsequent statements.

  The name of a statement entity can also be the name of a global or local entity in the same scoping unit; in this case, the name is interpreted within the statement as that of the statement entity.

Table 15.1. Scope of Program Entities

Entity	**Scope**	
Program units	Global	
Common blocks[1]	Global	
External procedures	Global	
Intrinsic procedures	Global[2]	
Module procedures	Local	Class I
Internal procedures	Local	Class I
Dummy procedures	Local	Class I
Statement functions	Local	Class I
Derived types	Local	Class I
Components of derived types	Local	Class II
Named constants	Local	Class I
Named constructs	Local	Class I
Namelist group names	Local	Class I
Generic identifiers	Local	Class I
Argument keywords in procedures	Local	Class III
Variables that can be referenced throughout a subprogram	Local	Class I
Variables that are dummy arguments in statement functions	Statement	

Entity	Scope	
DO variables in an implied-do list[3] of a DATA or FORALL statement, or an array constructor	Statement	
Intrinsic operators	Global	
Defined operators	Local	
Statement labels	Local	
External I/O unit numbers	Global	
Intrinsic assignment	Global[4]	
Defined assignment	Local	

[1]Names of common blocks can also be used to identify local entities.

[2]If an intrinsic procedure is not used in a scoping unit, its name can be used as a local entity within that scoping unit. For example, if intrinsic function COS is not used in a program unit, COS can be used as a local variable there.

[3]The DO variable in an implied-do list of an I/O list has local scope.

[4]The scope of the assignment symbol (=) is global, but it can identify additional operations (see Section 8.9.5).

Scoping units can contain other scoping units. For example, the following shows six scoping units:

```
MODULE MOD_1                        ! Scoping unit 1
   ...                              ! Scoping unit 1
CONTAINS                            ! Scoping unit 1
  FUNCTION FIRST                    ! Scoping unit 2
    TYPE NAME                       ! Scoping unit 3
    ...                             ! Scoping unit 3
    END TYPE NAME                   ! Scoping unit 3
    ...                             ! Scoping unit 2
  CONTAINS                          ! Scoping unit 2
    SUBROUTINE SUB_B                ! Scoping unit 4
      TYPE PROCESS                  ! Scoping unit 5
      ...                           ! Scoping unit 5
      END TYPE PROCESS              ! Scoping unit 5
      INTERFACE                     ! Scoping unit 5
        SUBROUTINE SUB_A            ! Scoping unit 6
        ...                         ! Scoping unit 6
        END SUBROUTINE SUB_A        ! Scoping unit 6
      END INTERFACE                 ! Scoping unit 5
    END SUBROUTINE SUB_B            ! Scoping unit 4
  END FUNCTION FIRST                ! Scoping unit 2
END MODULE                          ! Scoping unit 1
```

For More Information:

- On derived data types, see Section 3.3.
- On user-defined generic procedures, see Section 8.9.3.
- On intrinsic procedures, see Chapter 9.
- On procedures and subprograms, see Chapter 8.
- On use and host association, see Section 15.5.1.2.
- On defined operations, see Section 8.9.4.
- On defined assignment, see Section 8.9.5.

- On how the PRIVATE attribute can affect accessibility of entities, see Section 5.16.

15.3. Unambiguous Generic Procedure References

When a generic procedure reference is made, a specific procedure is invoked. If the following rules are used, the generic reference will be unambiguous:

- Within a scoping unit, two procedures that have the same generic name must both be subroutines (or both be functions). One of the procedures must have a nonoptional dummy argument that is one of the following:
  - Not present by position or argument keyword in the other argument list
  - Is present, but has different type and kind parameters, or rank
- Within a scoping unit, two procedures that have the same generic operator must both have the same number of arguments or both define assignment. One of the procedures must have a dummy argument that corresponds by position in the argument list to a dummy argument of the other procedure that has a different type and kind parameters, or rank.

When an interface block extends an intrinsic procedure, operator, or assignment, the rules apply as if the intrinsic consists of a collection of specific procedures, one for each allowed set of arguments.

When a generic procedure is accessed from a module, the rules apply to all the specific versions, even if some of them are inaccessible by their specific names.

For More Information:

For details on generic procedure names, see Section 8.9.3.

15.4. Resolving Procedure References

The procedure name in a procedure reference is either established to be generic or specific, or is not established. The rules for resolving a procedure reference differ depending on whether the procedure is established and how it is established.

15.4.1. References to Generic Names

Within a scoping unit, a procedure name is established to be generic if any of the following is true:

- The scoping unit contains an interface block with that procedure name.
- The procedure name matches the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is established to be generic in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be generic in a host scoping unit.

To resolve a reference to a procedure name established to be generic, the following rules are used in the order shown:

1. If an interface block with that procedure name appears in one of the following, the reference is to the specific procedure providing that interface:

   a. The scoping unit that contains the reference

   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with one of the specific interfaces of the interface block.

2. If the procedure name is specified with the INTRINSIC attribute in one of the following, the reference is to that intrinsic procedure:

   a. The same scoping unit

   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with the interface of that intrinsic procedure.

3. If the following is true, the reference is resolved by applying rules 1 and 2 to the host scoping unit:

   a. The procedure name is established to be generic in the host scoping unit

   b. There is agreement between the scoping unit and the host scoping unit as to whether the procedure is a function or subroutine name.

4. If none of the preceding rules apply, the reference must be to the generic intrinsic procedure with that name. The reference must be consistent with the interface of that intrinsic procedure.

15.4.2. References to Specific Names

In a scoping unit, a procedure name is established to be specific if it is not established to be generic and any of the following is true:

- The scoping unit contains an interface body with that procedure name.
- The scoping unit contains an internal procedure, module procedure, or statement function with that procedure name.
- The procedure name is the same as the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is specified with the EXTERNAL attribute in that scoping unit.
- The procedure name is established to be specific in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be specific in a host scoping unit.

To resolve a reference to a procedure name established to be specific, the following rules are used in the order shown:

1. If either of the following is true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram, and it contains an interface body with that procedure name.

   b. The procedure name has been declared EXTERNAL, and the procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.

2. If the scoping unit contains an interface body or the procedure name has been declared EXTERNAL, and Rule 1 does not apply, the reference is to an external procedure with that name.

3. If the scoping unit contains an internal procedure or statement function with that procedure name, the reference is to that entity.

4. If the procedure name has been declared INTRINSIC in the scoping unit, the reference is to the intrinsic procedure with that name.

5. If the scoping unit contains a USE statement that makes the name of a module procedure accessible, the reference is to that procedure. (The USE statement allows renaming, so the name referenced may differ from the name of the module procedure).

6. If none of the preceding rules apply, the reference is resolved by applying these rules to the host scoping unit.

15.4.3. References to Nonestablished Names

In a scoping unit, a procedure name is not established if it is not determined to be generic or specific.

To resolve a reference to a procedure name that is not established, the following rules are used in the order shown:

1. If both of the following are true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram.

   b. The procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.

2. If both of the following are true, the procedure is an intrinsic procedure and the reference is to that intrinsic procedure:

   a. The procedure name matches the name of an intrinsic procedure.

   b. There is agreement between the intrinsic procedure definition and the reference of the name as a function or subroutine.

3. If neither of the preceding rules apply, the reference is to an external procedure with that name.

For More Information:

- On subroutine references, see Section 7.3.
- On function references, see Section 8.5.2.2.

- On generic procedure names, see Section 8.9.3.
- On the USE statement, see Section 8.3.2.

15.5. Association

Association allows different program units to access the same value through different names. Entities are associated when each is associated with the same storage location.

There are three kinds of association:

- Name association (Section 15.5.1)
- Pointer association (Section 15.5.2)
- Storage association (Section 15.5.3)

Example 15.1 shows name, pointer, and storage association between an external program unit and an external procedure.

Example 15.1. Example of Name, Pointer, and Storage Association

```
! Scoping Unit 1: An external program unit

REAL A, B(4)
REAL, POINTER :: M(:)
REAL, TARGET :: N(12)
COMMON /COM/...
EQUIVALENCE (A, B(1))          ! Storage association between A and B(1)
M => N                         ! Pointer association
CALL P (actual-arg,...)
...

! Scoping Unit 2: An external procedure
SUBROUTINE P (dummy-arg,...)  ! Name and storage association between
                              !    these arguments and the calling
                              !    routine's arguments in scoping unit 1

  COMMON /COM/...             ! Storage association with common block COM
                              !    in scoping unit 1
  REAL Y
  CALL Q (actual-arg,...)
  CONTAINS
    SUBROUTINE Q (dummy-arg,...) ! Name and storage association between
                                 !   these arguments and the calling
                                 !   routine's arguments in host procedure
                                 !   P (subprogram Q has host association
                                 !   with procedure P)
      Y = 2.0*(Y-1.0)          ! Name association with Y in host procedure P
    ...
```

15.5.1. Name Association

Name association allows an entity (such as the name of a variable, constant, or procedure) to be accessed from different scoping units by the same name or by different names. There are three types of name association: argument, use, and host.

15.5.1.1. Argument Association

Arguments are the values passed to and from functions and subroutines through calling program argument lists.

Execution of a procedure reference establishes argument association between an actual argument and its corresponding dummy argument. The name of a dummy argument can be different from the name of its associated actual argument (if any).

When the procedure completes execution, the argument association is terminated.

For More Information:

For details on argument association, see Section 8.8.

15.5.1.2. Use and Host Association

Use association allows the entities in a module to be accessible to other scoping units. The mechanism for use association is the USE statement. The USE statement provides access to all public entities in the module, unless ONLY is specified. In this case, only the entities named in the ONLY list can be accessed.

Host association allows the entities in a host scoping unit to be accessible to an internal procedure, derived-type definition, or module procedure contained within the host. The accessed entities are known by the same name and have the same attributes as in the host. Entities that are local to a procedure are not accessible to its host.

Use or host association remains in effect throughout the execution of the executable program.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible. A name that appears in the scoping unit as an external name in an EXTERNAL statement is a global name, and any entity of the host that has this as its nongeneric name is inaccessible.

An interface body does not access named entities by host association, but it can access entities by use association.

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association can be changed within the procedure. After execution of the procedure, the pointer association remains current, unless the execution caused the target to become undefined. If this occurs, the host associated pointer becomes undefined.

Note

Implicit declarations can cause problems for host association. It is recommended that you use IMPLICIT NONE in both the host and the contained procedure, and that you explicitly declare all entities.

When all entities are explicitly declared, local declarations override host declarations, and host declarations that are not overridden are available in the contained procedure.

The following example shows host and use association:

```
MODULE SHARE_DATA
  REAL Y, Z
```

```
END MODULE

PROGRAM DEMO
  USE SHARE_DATA        ! All entities in SHARE_DATA are available
  REAL B, Q             !   through use association.
  ...
  CALL CONS (Y)
CONTAINS
  SUBROUTINE CONS (Y)   ! Y is a local entity (dummy argument).
    REAL C, Y
    ...
    Y = B + C + Q + Z   ! B and Q are available through host association.
    ...                 !   C is a local entity, explicitly declared.  Z
  END SUBROUTINE CONS   !   is available through use association.
END PROGRAM DEMO
```

For More Information:

- On the USE statement, see Section 8.3.2.
- On entities with local scope, see Section 15.2.

15.5.2. Pointer Association

A pointer can be associated with a target. At different times during the execution of a program, a pointer can be undefined, associated with different targets, or be disassociated. The initial association status of a pointer is undefined. A pointer can become associated by the following:

- By pointer assignment (pointer => target)

  The target must be associated, or specified with the TARGET attribute. If the target is allocatable, it must be currently allocated.

- By allocation (successful execution of an ALLOCATE statement)

  The ALLOCATE statement must reference the pointer.

A pointer becomes disassociated if any of the following occur:

- The pointer is nullified by a NULLIFY statement.
- The pointer is deallocated by a DEALLOCATE statement.
- The pointer is assigned a disassociated pointer (or the NULL intrinsic function).

When a pointer is associated with a target, the definition status of the pointer is defined or undefined, depending on the definition status of the target. A target is undefined in the following cases:

- If it was never allocated
- If it is not deallocated through the pointer
- If a RETURN or END statement causes it to become undefined

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 associated with a target. When a pointer is nullified, it is disassociated. When a pointer is allocated, it becomes associated, but is undefined. When a pointer is associated with a target, its association and definition status are determined by its target.

For More Information:

- On pointer assignment, see Section 4.2.3.
- On the ALLOCATE and DEALLOCATE statements, see Chapter 6.
- On the NULLIFY statement, see Chapter 6.
- On the NULL intrinsic function, see Section 9.4.110.

15.5.3. Storage Association

Storage association is the association of two or more data objects. It occurs when two or more storage sequences share (or are aligned with) one or more **storage units**. Storage sequences are used to describe relationships among variables, common blocks, and result variables.

15.5.3.1. Storage Units and Storage Sequence

A **storage unit** is a fixed unit of physical memory allocated to certain data. A **storage sequence** is a sequence of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit can be numeric, character, or unspecified.

A nonpointer scalar of type default real, integer, or logical occupies one numeric storage unit. A nonpointer scalar of type double precision real or default complex occupies two contiguous numeric storage units. In VSI Fortran, one numeric storage unit corresponds to 4 bytes of memory.

A nonpointer scalar of type default character with character length 1 occupies one character storage unit. A nonpointer scalar of type default character with character length *len* occupies *len* contiguous character storage units. In VSI Fortran, one character storage unit corresponds to 1 byte of memory.

A nonpointer scalar of nondefault data type occupies a single unspecified storage unit. The number of bytes corresponding to the unspecified storage unit differs depending on the data type.

Table 15.2 lists the storage requirements (in bytes) for the intrinsic data types.

Table 15.2. Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1
LOGICAL	2, 4, or 8[1]
LOGICAL(1)	1
LOGICAL(2)	2
LOGICAL(4)	4
LOGICAL(8)	8
INTEGER	2, 4, 8[1]

Data Type	Storage Requirements (in bytes)
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8
REAL	4, 8, or 16[2]
REAL(4)	4
DOUBLE PRECISION	8
REAL(8)	8
REAL (16)	16
COMPLEX	8, 16, or 32[2]
COMPLEX(4)	8
DOUBLE COMPLEX	16
COMPLEX(8)	16
COMPLEX (16)	32
CHARACTER	1
CHARACTER*len	len[3]
CHARACTER* (*)	assumed-length[4]

[1]Depending on default integer, LOGICAL and INTEGER can have 2, 4, or 8 bytes. The default allocation is four bytes.

[2]Depending on default real, REAL can have 4, 8, or 16 bytes and COMPLEX can have 8, 16, or 32 bytes. The default allocations are four bytes for REAL and eight bytes for COMPLEX.

[3]The value of len is the number of characters specified. The largest valid value is 65535. Negative values are treated as zero.

[4]The assumed-length format * (*) applies to dummy arguments, PARAMETER statements, or character functions, and indicates that the length of the actual argument or function is used. (See Section 8.8.4 and the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].)

A nonpointer scalar of sequence derived type occupies a sequence of storage sequences corresponding to the components of the structure, in the order they occur in the derived-type definition. (A sequence derived type has a SEQUENCE statement).

A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank.

The definition status and value of a data object affects the definition status and value of any storage-associated entity.

When two objects occupy the same storage sequence, they are totally storage-associated. When two objects occupy parts of the same storage sequence, they are partially associated. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement can cause total or partial storage association of storage sequences.

For More Information:

- On the COMMON statement, see Section 5.4.
- On the ENTRY statement, see Section 8.11.

- On the EQUIVALENCE statement, see Section 5.7.
- On the hardware representations of data types, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

15.5.3.2. Array Association

A nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order.

Two or more arrays are associated when each one is associated with the same storage location. They are partially associated when part of the storage associated with one array is the same as part or all of the storage associated with another array.

If arrays with different data types are associated (or partially associated) with the same storage location, and the value of one array is defined (for example, by assignment), the value of the other array becomes undefined. This happens because an element of an array is considered defined only if the storage associated with it contains data of the same type as the array name.

An array element, array section, or whole array is defined by a DATA statement before program execution. (The array properties must be declared in a previous specification statement.) During program execution, array elements and sections are defined by an assignment or input statement, and entire arrays are defined by input statements.

For More Information:

- On arrays, see Section 3.5.2.
- On array element order, see Section 3.5.2.2.
- On the DATA statement, see Section 5.5.

Appendix A. Deleted and Obsolescent Language Features

This appendix describes deleted and obsolescent language features.

Fortran 90 identified certain FORTRAN 77 features to be obsolescent. Fortran 95 deleted some of these features, and identified a few more language features to be obsolescent. Features considered obsolescent might be removed from future revisions of the Fortran Standard.

You can specify a compiler option to have these features flagged.

Note

VSI Fortran fully supports features deleted from Fortran 95.

A.1. Deleted Language Features in Fortran 95

Some language features, considered redundant in FORTRAN 77, are not included in Fortran 95. However, they are still fully supported by VSI Fortran:

- ASSIGN and assigned GO TO statements
- Assigned FORMAT specifier
- Branching to an END IF statement from outside its IF block
- H edit descriptor
- PAUSE statement
- Real and double precision DO control variables and DO loop control expressions

For suggested methods to achieve the functionality of these features, see Section A.3.

A.2. Obsolescent Language Features in Fortran 95

Some language features considered redundant in Fortran 90 are identified as obsolescent in Fortran 95.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate returns

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program use a CASE construct to test the value and perform operations (see Section 7.4).

- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct (see Section 7.8).

- Assumed-length character functions

  To replace this functionality, it is recommended that you use one of the following:

  - An automatic character-length function, where the length of the function result is declared in a specification expression
  - A subroutine whose arguments correspond to the function result and the function arguments

  Dummy arguments of a function can still have assumed character length; this feature is not obsolescent.

- CHARACTER*(*) form of CHARACTER declaration

  To replace this functionality, it is recommended that you use the Fortran 90 forms of specifying a length selector in CHARACTER declarations (see Section 5.1.2).

- Computed GO TO statement

  To replace this functionality, it is recommended that you use a CASE construct (see Section 7.4).

- DATA statements among executable statements

  This functionality has been included since FORTRAN 66, but is considered to be a potential source of errors.

- Fixed source form

  Newer methods of entering data have made this source form obsolescent and error-prone.

  The recommended method for coding is to use free source form (see Section 2.3.1).

- Shared DO termination and termination on a statement other than END DO or CONTINUE

  To replace this functionality, it is recommended that you use an END DO statement (see Section 7.6.1) or a CONTINUE statement (see Section 7.5).

- Statement functions

  To replace this functionality, it is recommended that you use an internal function (see Section 8.7).

A.3. Obsolescent Language Features in Fortran 90

Fortran 90 did not delete any of the features in FORTRAN 77, but some FORTRAN 77 features were identified as obsolescent.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate return (labels in an argument list)

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program test the value and perform operations, using a computed GO TO statement (see Section 7.2.2) or CASE construct (see Section 7.4).

- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct (see Section 7.8).

- ASSIGN and assigned GO TO statements

  These statements are usually used to simulate internal procedures (see Section 8.7), which can now be coded directly.

- Assigned FORMAT specifier (label of a FORMAT statement assigned to an integer variable)

  To replace this functionality, it is recommended that you use character expressions to define format specifications (see Section 11.2).

- Branching to an END IF statement from outside its IF block

  To replace this functionality, it is recommended that you branch to the statement following the END IF statement (see Section 7.8.1).

- H edit descriptor

  To replace this functionality, it is recommended that you use the character constant edit descriptor (see Section 11.5).

- PAUSE statement

  To replace this functionality, it is recommended that you use a READ statement that awaits input data (see Section 10.3).

- Real and double precision DO control variables and DO loop control expressions

  To replace this functionality, it is recommended that you use integer DO variables and expressions (see Section 7.6).

- Shared DO termination and termination on a statement other than END DO or CONTINUE

  To replace this functionality, it is recommended that you use an END DO statement (see Section 7.6.1) or a CONTINUE statement (see Section 7.5).

Appendix B. Additional Language Features

This appendix describes additional language features provided by VSI Fortran to facilitate compatibility with older versions of Fortran.

Note

These language features are particularly useful in porting older Fortran programs to Fortran 95/90. However, you should avoid using them in new programs, especially new programs for which portability to other Fortran 95/90 implementations is important.

B.1. DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of files with relative organization and associates them with a logical unit number. The DEFINE FILE statement is comparable to the OPEN statement. In situations where you can use the OPEN statement, OPEN is the preferable mechanism for creating and opening files.

The DEFINE FILE statement takes the following form:

```
DEFINE FILE u(m, n, U, asv) [,u(m, n, U, asv)] . . .
```

u

Is a scalar integer constant or variable that specifies the logical unit number.

m

Is a scalar integer constant or variable that specifies the number of records in the file.

n

Is a scalar integer constant or variable that specifies the length of each record in 16-bit words (2 bytes).

U

Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv

Is a scalar integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to *asv*. The *asv* must not be a dummy argument.

Rules and Behavior

The DEFINE FILE statement specifies that a file containing *m* fixed-length records, each composed of *n* 16-bit words, exists (or will exist) on the specified logical unit. The records in the file are numbered sequentially from 1 through *m*.

A DEFINE FILE statement does not itself open a file. However, the statement must be executed before the first direct access I/O statement referring to the specified file. The file is opened when the I/O statement is executed.

If this I/O statement is a WRITE statement, a direct access sequential file is opened, or created if necessary.

If the I/O statement is a READ or FIND statement, an existing file is opened, unless the specified file does not exist. If a file does not exist, an error occurs.

The DEFINE FILE statement establishes the variable *asv* as the associated variable of a file. At the end of each direct access I/O operation, the Fortran I/O system places in *asv* the record number of the record immediately following the one just read or written.

The associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or FIND statement). So, direct access I/O statements can perform sequential processing on the file by using the associated variable of the file as the record number specifier.

Examples

In the following example, the DEFINE FILE statement specifies that the logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is 48 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

```
DEFINE FILE 3(1000,48,U,NREC)
```

B.2. ENCODE and DECODE Statements

The ENCODE and DECODE statements translate data and transfer it between variables or arrays in internal storage. The ENCODE statement translates data from internal (binary) form to character form; the DECODE statement translates data from character to internal form. These statements are comparable to using internal files in formatted sequential WRITE and READ statements, respectively.

The ENCODE and DECODE statements take the following forms:

```
ENCODE (c,f,b [,IOSTAT=i-var] [,ERR=label]) [io-list]
DECODE (c,f,b [,IOSTAT=i-var] [,ERR=label]) [io-list]
```

c

Is a scalar integer expression. In the ENCODE statement, *c* is the number of characters (in bytes) to be translated to character form. In the DECODE statement, *c* is the number of characters to be translated to internal form.

f

Is a format identifier. An error occurs if more than one record is specified.

b

Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression.

In the ENCODE statement, *b* receives the characters after translation to external form. If less than *c* characters are received, the remaining character positions are filled with blank characters. In the DECODE statement, *b* contains the characters to be translated to internal form.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see Section 10.2.1.7).

`label`

Is the label of an executable statement that receives control if an error occurs.

`io-list`

Is an I/O list (see Section 10.2.2).

In the ENCODE statement, the list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Rules and Behavior

The number of characters that the ENCODE or DECODE statement can translate depends on the data type of *b*. For example, an INTEGER (2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Examples

In the following example, the DECODE statement translates the 12 characters in A to integer form (as specified by the FORMAT statement):

```
      DIMENSION K(3)
      CHARACTER*12 A,B
      DATA A/'123456789012'/
      DECODE(12,100,A) K
100   FORMAT(3I4)
      ENCODE(12,100,B) K(3), K(2), K(1)
```

The 12 characters are stored in array K:

```
K(1) = 1234
K(2) = 5678
```

```
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B:

```
B = '901256781234'
```

For More Information:

- On internal READ statements, see Section 10.3.4.
- On internal WRITE statements, see Section 10.5.4.

B.3. FIND Statement

The FIND statement positions a direct access file at a particular record and sets the associated variable of the file to that record number. It is comparable to a direct access READ statement with no I/O list, and it can open an existing file. No data transfer takes place.

The FIND statement takes one of the following forms:

```
FIND ([UNIT=]io-unit, REC=r [,ERR=label] [,IOSTAT=i-var])
FIND (io-unit'rec [,ERR=label] [,IOSTAT=i-var])
```

`io-unit`

Is a logical unit number. It must refer to a relative organization file (see Section 10.2.1.1).

`r`

Is the direct access record number. It cannot be less than one or greater than the number of records defined for the file (see Section 10.2.1.4).

`label`

Is the label of the executable statement that receives control if an error occurs.

`i-var`

Is a scalar integer variable that is defined as a positive integer if an error occurs, and as zero if no error occurs (see Section 10.2.1.7).

Examples

In the following example, the FIND statement positions logical unit 1 at the first record in the file. The file's associated variable is set to one:

```
FIND(1, REC=1)
```

In the following example, the FIND statement positions the file at the record identified by the content of INDX. The file's associated variable is set to the value of INDX:

```
FIND(4, REC=INDX)
```

For More Information:

On direct access READ statements, see Section 10.3.2.

B.4. FORTRAN-66 Interpretation of the EXTERNAL Statement

If you specify the compiler option indicating FORTRAN-66 semantics, the EXTERNAL statement is interpreted in a way that was specified by the FORTRAN IV (FORTRAN-66) standard. This interpretation became incompatible with FORTRAN 77 and later revisions of the Fortran standard.

The FORTRAN-66 interpretation of the EXTERNAL statement combines the functionality of the INTRINSIC statement (Section 5.11) with that of the EXTERNAL statement (Section 5.8).

This lets you use subprograms as arguments to other subprograms. The subprograms to be used as arguments can be either user-supplied functions or Fortran 95/90 library functions.

The FORTRAN-66 EXTERNAL statement takes the following form:

```
EXTERNAL [*]v [,[*]v] . . .
```

*****

Specifies that a user-supplied function is to be used instead of a Fortran 95/90 library function having the same name.

v

Is the name of a subprogram or the name of a dummy argument associated with the name of a subprogram.

Rules and Behavior

The FORTRAN-66 EXTERNAL statement declares that each name in its list is an external function name. Such a name can then be used as an actual argument to a subprogram, which then can use the corresponding dummy argument in a function reference or CALL statement.

However, when used as an argument, a complete function reference represents a value, not a subprogram name; for example, SQRT(B) in CALL SUBR(A, SQRT(B), C). It is not, therefore, defined in an EXTERNAL statement (as would be the incomplete reference SQRT).

Examples

Example B.1 demonstrates the FORTRAN-66 EXTERNAL statement.

Example B.1. Using the F66 EXTERNAL Statement

```
Main Program                          Subprograms

EXTERNAL SIN, COS, *TAN, SINDEG       SUBROUTINE TRIG(X,F,Y)
   .                                  Y = F(X)
```

```
   .                                  RETURN
   .                                  END
CALL TRIG(ANGLE, SIN, SINE)
   .
   .                                  FUNCTION TAN(X)
   .                                  TAN = SIN(X)/COS(X)
CALL TRIG(ANGLE, COS, COSINE)         RETURN
   .                                  END
   .
   .
CALL TRIG(ANGLE, TAN, TANGNT)         FUNCTION SINDEG(X)
   .                                  SINDEG = SIN(X*3.1459/180)
   .                                  RETURN
   .                                  END
CALL TRIG(ANGLED, SINDEG, SINE)
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)
```

The functions SIN and COS are examples of trigonometric functions supplied in the Fortran 95/90 library. The function TAN is also supplied in the library, but the asterisk (*) in the EXTERNAL statement specifies that the user-supplied function be used, instead of the library function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

For More Information:

On Fortran 95/90 intrinsic functions, see Chapter 9.

B.5. Alternative Syntax for the PARAMETER Statement

The PARAMETER statement discussed here is similar to the one discussed in Section 5.14; they both assign a name to a constant. However, this PARAMETER statement differs from the other one in the following ways:

- Its list is not bounded with parentheses.
- The form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

This PARAMETER statement takes the following form:

```
PARAMETER c = expr [,c = expr]...
```

c

Is the name of the constant.

expr

Is an initialization expression. It can be of any data type.

Rules and Behavior

Each name *c* becomes a constant and is defined as the value of expression *expr*. Once a name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the name.

The name of a constant cannot appear as part of another constant, except as the real or imaginary part of a complex constant. For example:

```
PARAMETER I=3
PARAMETER M=I.25        ! Not allowed
PARAMETER N=(1.703, I)  ! Allowed
```

The name used in the PARAMETER statement identifies only the name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The name of a constant assumes the data type of its corresponding constant expression. The data type of a parameter constant cannot be specified in a type declaration statement. Nor does the initial letter of the constant's name implicitly affect its data type.

Examples

The following are valid examples of this form of the PARAMETER statement:

```
PARAMETER  PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER  PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER  FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

For More Information:

On compile-time constant expressions, see Section 5.14.

B.6. VIRTUAL Statement

The VIRTUAL statement is included for compatibility with PDP-11 Fortran. It has the same form and effect as the DIMENSION statement (see Section 5.6).

B.7. Alternative Syntax for Octal and Hexadecimal Constants

In VSI Fortran, you can use an alternative syntax for octal and hexadecimal constants. The following table shows this alternative syntax and equivalents:

Constant	Alternative Syntax	Equivalent
Octal	'0..7'O	O'0..7'

Constant	Alternative Syntax	Equivalent
Hexadecimal	'0..F'X	Z'0..F'

You can use a quotation mark (") in place of an apostrophe in all the above syntax forms.

For More Information:

- On octal constants, see Section 3.4.2.
- On hexadecimal constants, see Section 3.4.3.

B.8. Alternative Syntax for a Record Specifier

In VSI Fortran, you can specify the following form for a record specifier in an I/O control list:

```
’r
```

r

Is a numeric expression with a value that represents the position of the record to be accessed using direct access I/O.

The value must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file. If necessary, a record number is converted to integer data type before being used.

If this nonkeyword form is used in an I/O control list, it must immediately follow the nonkeyword form of the io-unit specifier.

B.9. Alternative Syntax for the DELETE Statement

In VSI Fortran, you can specify the following form of the DELETE statement when deleting records from a relative file:

```
DELETE (io-unit’r [,ERR=label] [,IOSTAT=i-var])
```

io-unit

Is the number of the logical unit containing the record to be deleted.

r

Is the positional number of the record to be deleted.

label

Is the label of an executable statement that receives control if an error condition occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

This form deletes the direct access record specified by *r*.

For More Information:

On the DELETE statement, see Section 12.3.

B.10. Alternative Form for Namelist External Records

In VSI Fortran, you can use the following form for an external record:

```
$group-name object = value [object = value]...$[END]
```

group-name

Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit.

object

Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks, but it can be preceded or followed by blanks.

value

Is a null value, a constant (or list of constants), a repetition of constants in the form r*c, or a repetition of null values in the form r*.

If more than one *object=value* or more than one value is specified, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks.

For More Information:

On namelist input, see Section 10.3.1.3; output, see Section 10.5.1.3.

B.11. VSI Fortran POINTER Statement

The POINTER statement discussed here is different from the one discussed in Section 5.15. It establishes pairs of variables and pointers, in which each pointer contains the address of its paired variable.

This POINTER statement takes the following form:

```
POINTER (pointer,pointee) [,(pointer,pointee)] . . .
```

pointer

Is a variable whose value is used as the address of the pointee.

pointee

Is a variable; it can be an array name or array specification.

Rules and Behavior

The following are *pointer* rules and behavior:

- Two pointers can have the same value, so pointer aliasing is allowed.
- When used directly, a pointer is treated like an integer variable. A pointer occupies two numeric storage units, so it is a 64-bit quantity (INTEGER(8)).
- A pointer cannot be a pointee.
- A pointer cannot appear in an ASSIGN statement and cannot have the following attributes:

ALLOCATABLE	INTRINSIC	POINTER
EXTERNAL	PARAMETER	TARGET

  A pointer can appear in a DATA statement with integer literals only.
- Integers can be converted to pointers, so you can point to absolute memory locations.
- A pointer variable cannot be declared to have any other data type.
- A pointer cannot be a function return value.
- You can give values to pointers by doing the following:
  - Retrieve addresses by using the LOC intrinsic function (or the %LOC built-in function)
  - Allocate storage for an object by using the MALLOC intrinsic function (or by using LIB $GET_VM)

  For example:

Using %LOC:

```
INTEGER I(10)
INTEGER I1(10) /10*10/
POINTER (P,I)
P = %LOC(I1)
I(2) = I(2) + 1
```

Using MALLOC:

```
INTEGER I(10)
POINTER (P,I)
P = MALLOC(40)
I(2) = I(2) + 1
```

Using LIB$GET_VM:

```
INTEGER I(10)
INTEGER LIB$GET_VM,STATUS
POINTER (P,I)
STATUS = LIB$GET_VM(P,40)
IF (.NOT. STATUS)
        CALL EXIT(STATUS)
I(2) = I(2) + 1
```

- The value in a pointer is used as the pointee's base address.

The following are *pointee* rules and behavior:

- A pointee is not allocated any storage. References to a pointee look to the current contents of its associated pointer to find the pointee's base address.

- A pointee cannot be data-initialized or have a record structure that contains data-initialized fields.
- A pointee can appear in only one POINTER statement.
- A pointee array can have fixed, adjustable, or assumed dimensions.
- A pointee cannot appear in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement, and it cannot have the following attributes:

ALLOCATABLE	OPTIONAL	SAVE
AUTOMATIC	PARAMETER	STATIC
INTENT	POINTER	TARGET

- A pointee cannot be:
  - A dummy argument
  - A function return value
  - A record field or an array element
  - Zero-sized
  - An automatic object
  - The name of a generic interface block
- If a pointee is of derived type, it must be of sequence type.

B.12. Record Structures

VSI Fortran record structures are similar to Fortran 95/90 derived types.

A **record structure** is an aggregate entity containing one or more elements. (Record elements are also called fields or components.) You can use records when you need to declare and operate on multi-field data structures in your programs.

Creating a record is a two-step process:

1. You must define the form of the record with a multistatement **structure declaration**.
2. You must use a RECORD statement to declare the record as an entity with a name. (More than one RECORD statement can refer to a given structure).

For More Information:

On derived types, see Section 3.3.

B.12.1. Structure Declarations

A structure declaration defines the field names, types of data within fields, and order and alignment of fields within a record. Fields and structures can be initialized, but records cannot be initialized.

A structure declaration takes the following form:

```
STRUCTURE [/structure-name/][field-namelist]
field-declaration
[field-declaration]
. . .
[field-declaration]
END STRUCTURE
```

structure-name

Is the name used to identify a structure, enclosed by slashes.

Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names, but structures can share names with variables (scalar or array), record fields, PARAMETER constants, and common blocks.

Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and is optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name.

Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structures within the calling and called subprograms must match in type, order, and dimension.

field-namelist

Is a list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations.

field-declaration

Also called the declaration body. A *field-declaration* consists of any combination of the following:

- Type declarations (Section B.12.1.1)

  These are ordinary Fortran data type declarations.

- Substructure declarations (Section B.12.1.2)

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.

- Union declarations (Section B.12.1.3)

  A union declaration is composed of one or more mapped field declarations.

- PARAMETER statements

  PARAMETER statements can appear in a structure declaration, but cannot be given a data type within the declaration block.

Type declarations for PARAMETER names must precede the PARAMETER statement and be outside of a STRUCTURE declaration, as follows:

```
  INTEGER*4 P
STRUCTURE /ABC/
  PARAMETER (P=4)
  REAL*4 F
END STRUCTURE
  REAL*4 A(P)
```

Rules and Behavior

Unlike type declaration statements, structure declarations do not create variables. Structured variables (records) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type declaration statement. The difference is that aggregate items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important, because this ordering determines the order of the fields in records.

In a structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

By default, fields are aligned on natural boundaries; misaligned fields are padded as necessary. To avoid padding of records, you should lay out structures so that all fields are naturally aligned.

To pack fields on arbitrary byte boundaries, you must specify a compiler option. You can also specify alignment for fields by using the cDEC$ OPTIONS or cDEC$ PACK general directive.

A field name must not be the same as any intrinsic or user-defined operator (for example, EQ cannot be used as a field name).

Examples

In the following example, the declaration defines a structure named APPOINTMENT. APPOINTMENT contains the structure DATE (field APP_DATE) as a substructure. It also contains a substructure named TIME (field APP_TIME, an array), a CHARACTER*20 array named APP_MEMO, and a LOGICAL*1 field named APP_FLAG.

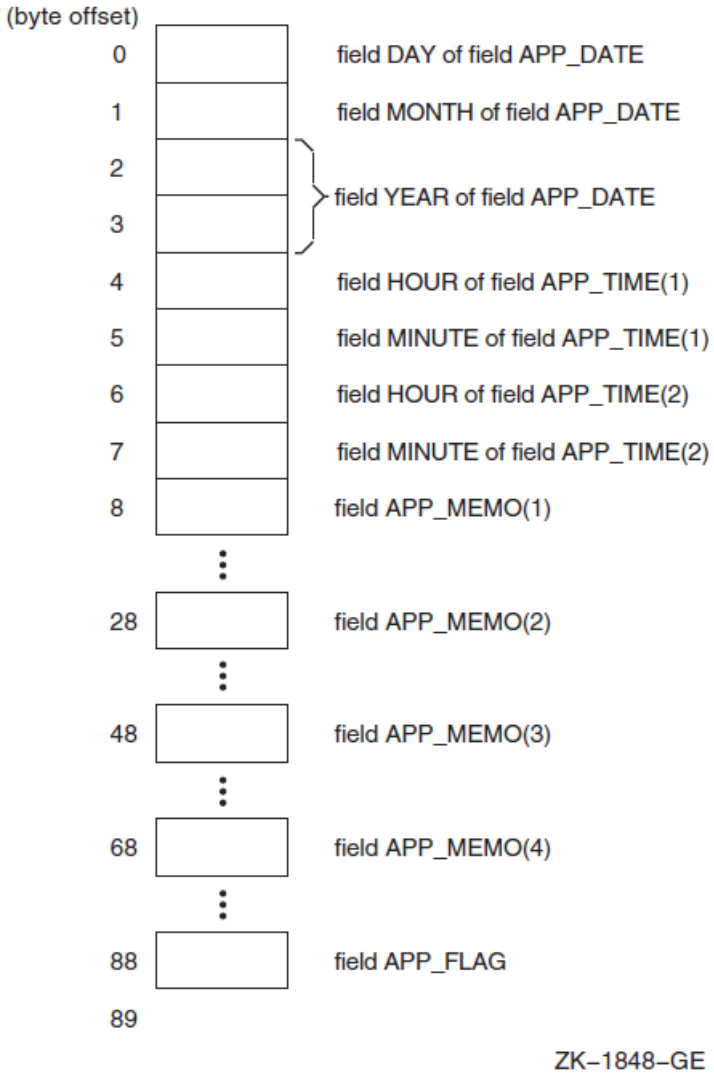
```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

STRUCTURE /APPOINTMENT/
  RECORD /DATE/     APP_DATE
  STRUCTURE /TIME/  APP_TIME (2)
    INTEGER*1       HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20      APP_MEMO (4)
  LOGICAL*1         APP_FLAG
END STRUCTURE
```

The length of any instance of structure APPOINTMENT is 89 bytes.

Figure B.1 shows the memory mapping of any record or record array element with the structure APPOINTMENT.

Figure B.1. Memory Map of Structure APPOINTMENT



For More Information:

- On compiler options, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].
- On the cDEC$ OPTIONS directive, see Section 14.14.

B.12.1.1. Type Declarations

The syntax of a type declaration within a record structure is identical to that of a normal Fortran type statement.

The following rules and behavior apply to type declarations in record structures:

- %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment. This creates an unnamed field.

%FILL can have an array specification; for example:

```
INTEGER %FILL (2,2)
```

Unnamed fields cannot be initialized. For example, the following statement is invalid and generates an error message:

```
INTEGER %FILL /1980/
```

- Initial values can be supplied in field declaration statements. Unnamed fields cannot be initialized; they are always undefined.
- Field names must always be given explicit data types. The IMPLICIT statement does not affect field declarations.
- Any required array dimensions must be specified in the field declaration statements. DIMENSION statements cannot be used to define field names.
- Adjustable or assumed sized arrays and assumed-length CHARACTER declarations are not allowed in field declarations.

B.12.1.2. Substructure Declarations

A field within a structure can itself be a structured item composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations (with the limitation that you cannot refer to a structure inside itself at any level of nesting).

  One or more field names must be defined in the STRUCTURE statement for the substructure, because all fields in a structure must be named. In this case, the substructure is being used as a field within a structure or union.

  Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict.
- By using a RECORD statement that specifies another previously defined record structure, thereby including it in the structure being declared.

See the example in Section B.12.1 for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

B.12.1.3. Union Declarations

A union declaration is a multistatement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a structure declaration.

Each unique field or group of fields is defined by a separate map declaration.

A union declaration takes the following form:

```
UNION
   map-declaration
   map-declaration
   [map-declaration]
   . . .
   [map-declaration]
END UNION
```

map-declaration

Takes the following form:

```
MAP
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END MAP
```

field-declaration

Is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a data field (having a data type) within a union. See Section B.12.1 for a more detailed description of what can be specified in field declarations.

Rules and Behavior

As with normal Fortran type declarations, data can be initialized in field declaration statements in union declarations. However, if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the fields declared within it.

Manipulating data by using union declarations is similar to using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are concurrently associated with a common storage location and the data residing there; with union declarations you can use one discrete storage location to alternately contain a variety of fields (arrays or variables).

With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

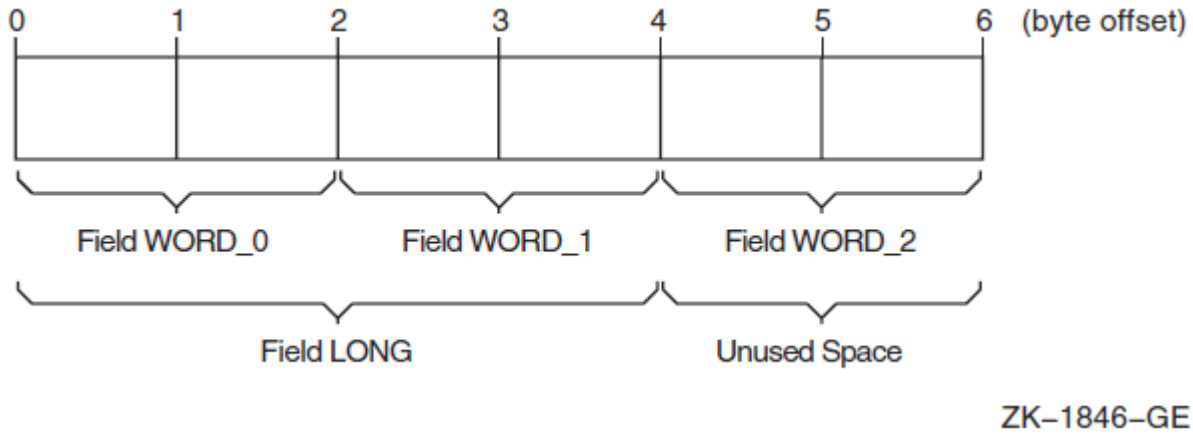
Examples

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG:

```
STRUCTURE /WORDS_LONG/
  UNION
    MAP
      INTEGER*2  WORD_0, WORD_1, WORD_2
    END MAP
    MAP
      INTEGER*4  LONG
    END MAP
  END UNION
END STRUCTURE
```

The length of any record with the structure WORDS_LONG is 6 bytes. Figure B.2 shows the memory mapping of any record with the structure WORDS_LONG:

Figure B.2. Memory Map of Structure WORDS_LONG



B.12.2. RECORD Statement

A RECORD statement takes the following form:

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

`structure-name`

Is the name of a previously declared structure.

`record-namelist`

Is a list of one or more variable names, array names, or array specifications, separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

Rules and Behavior

You can use record names in COMMON and DIMENSION statements, but not in DATA or NAMELIST statements.

Records initially have undefined values unless you have defined their values in structure declarations.

B.12.3. References to Record Fields

References to record fields must correspond to the kind of field being referenced. Aggregate field references refer to composite structures (and substructures). Scalar field references refer to singular data items, such as variables.

An operation on a record can involve one or more fields.

Record field references take one of the following forms:

```
record-name [.aggregate-field-name] . . .

record-name [.aggregate-field-name] . . . .scalar-field-name
```

record-name

Is the name used in a RECORD statement to identify a record.

aggregate-field-name

Is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure identified by the record name.

scalar-field-name

Is the name of a data item (having a data type) defined within a structure declaration.

Rules and Behavior

Records and record fields cannot be used in DATA statements, but individual fields can be initialized in the STRUCTURE definition.

An automatic array cannot be a record field.

A scalar field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names followed by the name of a scalar field. A scalar field reference refers to a single data item (having a data type) and can be treated like a normal reference to a Fortran variable or array element.

You can use scalar field references in statement functions and in executable statements. However, they cannot be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements, or as the control variable in an indexed DO-loop.

Type conversion rules for scalar field references are the same as those for variables and array elements.

An aggregate field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names.

You can only assign an aggregate field to another aggregate field (record = record) if the records have the same structure. VSI Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

VSI Fortran requires qualification on all levels. While some languages allow omission of aggregate field names when there is no ambiguity as to which field is intended, VSI Fortran requires all aggregate field names to be included in references.

You can use aggregate field references in unformatted I/O statements; one I/O record is written no matter how many aggregate and array name references appear in the I/O list. You cannot use aggregate field references in formatted, namelist, and list-directed I/O statements.

You can use aggregate field references as actual arguments and record dummy arguments. The declaration of the dummy record in the subprogram must match the form of the aggregate field reference passed by the calling program unit; each structure must have the same number and types of fields in the same order. The order of map fields within a union declaration is irrelevant.

Records are passed by reference. Aggregate field references are treated like normal variables. You can use adjustable arrays in RECORD statements that are used as dummy arguments.

Note

Because periods are used in record references to separate fields, you should not use relational operators (.EQ., .XOR.), logical constants (.TRUE., .FALSE.), and logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Examples

The following examples show record and field references. Consider the following structure declarations:

Structure DATE:

```
STRUCTURE /DATE/
  INTEGER*1  DAY, MONTH
  INTEGER*2  YEAR
END STRUCTURE
```

Structure APPOINTMENT:

```
STRUCTURE /APPOINTMENT/
  RECORD /DATE/     APP_DATE
  STRUCTURE /TIME/  APP_TIME(2)
    INTEGER*1       HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20      APP_MEMO(4)
  LOGICAL*1         APP_FLAG
END STRUCTURE
```

The following RECORD statement creates a variable named NEXT_APP and a 10-element array named APP_LIST. Both the variable and each element of the array take the form of the structure APPOINTMENT.

```
RECORD /APPOINTMENT/ NEXT_APP,APP_LIST(10)
```

Each of the following examples of record and field references are derived from the previous structure declarations and RECORD statement:

Aggregate Field References

- The record NEXT_APP:

  ```
  NEXT_APP
  ```

- The field APP_DATE, a 4-byte array field in the record array APP_LIST (3):

```
APP_LIST(3).APP_DATE
```

Scalar Field References

- The field APP_FLAG, a LOGICAL field of the record NEXT_APP:

```
NEXT_APP.APP_FLAG
```

- The first character of APP_MEMO(1), a CHARACTER*20 field of the record NEXT_APP:

```
NEXT_APP.APP_MEMO(1)(1:1)
```

For More Information:

- On specification of fields within structure declarations, see Section B.12.1.
- On structure declarations, see Section B.12.1.
- On UNION and MAP statements, see Section B.12.1.3.
- On the RECORD statement, see Section B.12.2.
- On alignment of data, see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

B.12.4. Aggregate Assignment

For aggregate assignment statements, the variable and expression must have the same structure as the aggregate they reference.

The aggregate assignment statement assigns the value of each field of the aggregate on the right of an equal sign to the corresponding field of the aggregate on the left. Both aggregates must be declared with the same structure.

Examples

The following example shows valid aggregate assignments:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

RECORD /DATE/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
  ...
  RECORD /DATE/ APP_DATE
END STRUCTURE

RECORD /APPOINTMENT/ MEETING

DO I = 1,7
  CALL GET_DATE (TODAY)
  THIS_WEEK(I) = TODAY
  THIS_WEEK(I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE = TODAY
```

Appendix C. ASCII and DEC Multinational Character Sets

This appendix describes the ASCII and DEC Multinational character sets that are available on OpenVMS systems.

For details on the Fortran 95/90 character set, see Section 2.2.

C.1. ASCII Character Set

Figure C.1 represents the ASCII character set (characters with decimal values 0 through 127). The first half of each of the numbered columns identifies the character as you would enter it on a terminal or as you would see it on a printer. Except for SP and HT, the characters with names are nonprintable. In Figure C.1, the characters with names are defined as follows:

NUL	Null	DC1	Device Control 1 (XON)
SOH	Start of Heading	DC2	Device Control 2
STX	Start of Text	DC3	Device Control 3 (XOFF)
ETX	End of Text	DC4	Device Control 4
EOT	End of Transmission	NAK	Negative Acknowledge
ENQ	Enquiry	SYN	Synchronous Idle
ACK	Acknowledge	ETB	End of Transmission Block
BEL	Bell	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Horizontal Tab	SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tab	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	SP	Space
DLE	Data Link Escape	DEL	Delete

The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the uppercase letter A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

Figure C.1. Graphic Representation of the ASCII Character Set

Column	0		1		2		3		4		5		6		7	
Bits b8 b7 b6 b5 / Row b4 b3 b2 b1	0 0 0 0		0 0 0 1		0 0 1 0		0 0 1 1		0 1 0 0		0 1 0 1		0 1 1 0		0 1 1 1	
0 (0 0 0 0)	NUL	0 0 0	DLE	20 16 10	SP	40 32 20	0	60 48 30	@	100 64 40	P	120 80 50	`	140 96 60	p	160 112 70
1 (0 0 0 1)	SOH	1 1 1	DC1 (XON)	21 17 11	!	41 33 21	1	61 49 31	A	101 65 41	Q	121 81 51	a	141 97 61	q	161 113 71
2 (0 0 1 0)	STX	2 2 2	DC2	22 18 12	"	42 34 22	2	62 50 32	B	102 66 42	R	122 82 52	b	142 98 62	r	162 114 72
3 (0 0 1 1)	ETX	3 3 3	DC3 (XOFF)	23 19 13	#	43 35 23	3	63 51 33	C	103 67 43	S	123 83 53	c	143 99 63	s	163 115 73
4 (0 1 0 0)	EOT	4 4 4	DC4	24 20 14	$	44 36 24	4	64 52 34	D	104 68 44	T	124 84 54	d	144 100 64	t	164 116 74
5 (0 1 0 1)	ENQ	5 5 5	NAK	25 21 15	%	45 37 25	5	65 53 35	E	105 69 45	U	125 85 55	e	145 101 65	u	165 117 75
6 (0 1 1 0)	ACK	6 6 6	SYN	26 22 16	&	46 38 26	6	66 54 36	F	106 70 46	V	126 86 56	f	146 102 66	v	166 118 76
7 (0 1 1 1)	BEL	7 7 7	ETB	27 23 17	'	47 39 27	7	67 55 37	G	107 71 47	W	127 87 57	g	147 103 67	w	167 119 77
8 (1 0 0 0)	BS	10 8 8	CAN	30 24 18	(	50 40 28	8	70 56 38	H	110 72 48	X	130 88 58	h	150 104 68	x	170 120 78
9 (1 0 0 1)	HT	11 9 9	EM	31 25 19	)	51 41 29	9	71 57 39	I	111 73 49	Y	131 89 59	i	151 105 69	y	171 121 79
10 (1 0 1 0)	LF	12 10 A	SUB	32 26 1A	*	52 42 2A	:	72 58 3A	J	112 74 4A	Z	132 90 5A	j	152 106 6A	z	172 122 7A
11 (1 0 1 1)	VT	13 11 B	ESC	33 27 1B	+	53 43 2B	;	73 59 3B	K	113 75 4B	[	133 91 5B	k	153 107 6B	{	173 123 7B
12 (1 1 0 0)	FF	14 12 C	FS	34 28 1C	,	54 44 2C	<	74 60 3C	L	114 76 4C	\	134 92 5C	l	154 108 6C	|	174 124 7C
13 (1 1 0 1)	CR	15 13 D	GS	35 29 1D	-	56 45 2D	=	75 61 3D	M	115 77 4D	]	135 93 5D	m	155 109 6D	}	175 125 7D
14 (1 1 1 0)	SO	16 14 E	RS	36 30 1E	.	56 46 2E	>	76 62 3E	N	116 78 4E	^	136 94 5E	n	156 110 6E	~	176 126 7E
15 (1 1 1 1)	SI	17 15 F	US	37 31 1F	/	57 47 2F	?	77 63 3F	O	117 79 4F	_	137 95 5F	o	157 111 6F	DEL	177 127 7F

Key

Character: ESC | 33 27 1B (Octal Decimal Hex)

ZK-1752-GE

C.2. DEC Multinational Character Set

The ASCII character set comprises the first half of the DEC Multinational Character Set. Figure C.2 represents the second half of the DEC Multinational Character Set (characters with decimal values 128 through 255). The first half of each of the numbered columns identifies the character as you would see it on a terminal or printer (these characters cannot be output on some older terminals and printers). The characters with names are nonprintable. In Figure C.2, the characters with names are defined as follows:

IND	Index	PU1	Private Use 1
NEL	Next Line	PU2	Private Use 2
SSA	Start of Selected Area	STS	Set Transmit State
ESA	End of Selected Area	CCH	Cancel Character
HTS	Horizontal Tab Set	MW	Message Waiting
HTJ	Horizontal Tab Set with Justification	SPA	Start of Protected Area
VTS	Vertical Tab Set	EPA	End of Protected Area
PLD	Partial Line Down	CSI	Control Sequence Introducer
PLU	Partial Line Up	ST	String Terminator
RI	Reverse Index	OSC	Operating System Command
SS2	Single Shift 2	PM	Privacy Message
SS3	Single Shift 3	APC	Application
DCS	Device Control String		

The shaded boxes in Figure C.2 indicate positions that are not part of the character set.

Figure C.2. Graphic Representation of the DEC Multinational Extension to the ASCII Character Set

8		9		10		11		12		13		14		15		Column	
1 0 0 0		1 0 0 1		1 0 1 0		1 0 1 1		1 1 0 0		1 1 0 1		1 1 1 0		1 1 1 1		b8 b7 b6 b5 Bits b4 b3 b2 b1	Row
	200 128 80	DCS	220 144 90		240 160 A0	°	260 176 B0	À	300 192 C0		320 208 D0	à	340 224 E0		360 240 F0	0 0 0 0	0
	201 129 81	PU1	221 145 91	¡	241 161 A1	±	261 177 B1	Á	301 193 C1	Ñ	321 209 D1	á	341 225 E1	ñ	361 241 F1	0 0 0 1	1
	202 130 82	PU2	222 146 92	¢	242 162 A2	²	262 178 B2	Â	302 194 C2	Ò	322 210 D2	â	342 226 E2	ò	362 242 F2	0 0 1 0	2
	203 131 83	STS	223 147 93	£	243 163 A3	³	263 179 B3	Ã	303 195 C3	Ó	323 211 D3	ã	343 227 E3	ó	363 243 F3	0 0 1 1	3
IND	204 132 84	CCH	224 148 94		244 164 A4		264 180 B4	Ä	304 196 C4	Ô	324 212 D4	ä	344 228 E4	ô	364 244 F4	0 1 0 0	4
NEL	205 133 85	MW	225 149 95	¥	245 165 A5	µ	265 181 B5	Å	305 197 C5	Õ	325 213 D5	å	345 229 E5	õ	365 245 F5	0 1 0 1	5
SSA	206 134 86	SPA	226 150 96		246 166 A6	¶	266 182 B6	Æ	306 198 C6	Ö	326 214 D6	æ	346 230 E6	ö	366 246 F6	0 1 1 0	6
ESA	207 135 87	EPA	227 151 97	§	247 167 A7	·	267 183 B7	Ç	307 199 C7	Œ	327 215 D7	ç	347 231 E7	œ	367 247 F7	0 1 1 1	7
HTS	210 136 88		230 152 98	¤	250 168 A8		270 184 B8	È	310 200 C8	Ø	330 216 D8	è	350 232 E8	ø	370 248 F8	1 0 0 0	8
HTJ	211 137 89		231 153 99	©	251 169 A9	¹	271 185 B9	É	311 201 C9	Ù	331 217 D9	é	351 233 E9	ù	371 249 F9	1 0 0 1	9
VTS	212 138 8A		232 154 9A	ª	252 170 AA	º	272 186 BA	Ê	312 302 CA	Ú	332 218 DA	ê	352 234 EA	ú	372 250 FA	1 0 1 0	10
PLD	213 139 8B	CSI	233 155 9B	«	253 171 AB	»	273 187 BB	Ë	313 203 CB	Û	333 219 DB	ë	353 235 EB	û	373 251 FB	1 0 1 1	11
PLU	214 140 8C	ST	234 156 9C		254 172 AC	¼	274 188 BC	Ì	314 204 CC	Ü	334 220 DC	ì	354 236 EC	ü	374 252 FC	1 1 0 0	12
RI	215 141 8D	OSC	235 157 9D		255 173 AD	½	275 189 BD	Í	315 205 CD	Ÿ	335 221 DD	í	355 237 ED	ÿ	375 253 FD	1 1 0 1	13
SS2	216 142 8E	PM	236 158 9E		256 174 AE		276 190 BE	Î	316 206 CE		336 222 DE	î	356 238 EE		376 254 FE	1 1 1 0	14
SS3	217 143 8F	APC	237 159 9F		257 175 AF	¿	277 191 BF	Ï	317 207 CF	ß	337 223 DF	ï	357 239 EF		377 255 FF	1 1 1 1	15

Key

Character | ESC | 33 / 27 / 1B | Octal / Decimal / Hex

ZK-1753-GE

Appendix D. Data Representation Models

Several of the numeric intrinsic functions are defined by a model set for integers (for each integer kind used) and reals (for each real kind used). The bit functions are defined by a model set for bits (binary digits). This appendix describes these models.

For more information on the range of values for each data type (and kind), see the *VSI Fortran User Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-user-manual/].

D.1. Model for Integer Data

In general, the model set for integers is defined as follows:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

The following values apply to this model set:

- i is the integer value.
- s is the sign (either +1 or –1).
- q is the number of digits (a positive integer).
- r is the radix (an integer greater than 1).
- w_k is a nonnegative number less than r.

The model for INTEGER(4) is as follows:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

The following example shows the general integer model for $i = -20$ using a base (r) of 2:

$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$
$i = (-1) \times (4 + 16)$
$i = -1 \times 20$
$i = -20$

D.2. Model for Real Data

The model set for reals, in general, is defined as one of the following:

$$x = 0$$

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

The following values apply to this model set:

- x is the real value.
- s is the sign (either +1 or –1).
- b is the base (real radix; an integer greater than 1).
- p is the number of mantissa digits (an integer greater than 1). The number of digits differs depending on the real format, as follows:

IEEE S_floating	24
IEEE T_floating	53

- e is an integer in the range e_{min} to e_{max}, inclusive. This range differs depending on the real format, as follows:

	$\mathbf{e_{min}}$	$\mathbf{e_{max}}$
IEEE S_floating	–125	128
IEEE T_floating	–1021	1024

- f_k is a nonnegative number less than b (f_1 is also nonzero).

For x = 0, its exponent e and digits f_k are defined to be zero.

The model set for single-precision real (REAL(4)) is defined as one of the following:

$$x = 0$$

$$x = s \times 2^{e} \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], \; -125 \le e \le 128$$

The following example shows the general real model for x = 20.0 using a base (b) of 2:

$x = 1 \times 2^5 \times (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$
$x = 1 \times 32 \times (.5 + .125)$
$x = 32 \times (.625)$
$x = 20.0$

D.3. Model for Bit Data

The model set for bits (binary digits) interprets a nonnegative scalar data object of type integer as a sequence, as follows:

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

The following values apply to this model set:

- j is the integer value.
- s is the number of bits.
- w_k is a bit value of 0 or 1.

The bits are numbered from right to left beginning with 0.

The following example shows the bit model for *j* = 1001 (integer 9) using a bit number (*s*) of 4:

1	0	0	1
↓	↓	↓	↓
w_3	w_2	w_1	w_0

$j = (w_0 \times 2^0) + (w_1 \times 2^1) + (w_2 \times 2^2) + (w_3 \times 2^3)$
$j = 1 + 0 + 0 + 8$
$j = 9$

Appendix E. Summary of Language Extensions

This appendix summarizes the VSI Fortran language extensions to the ANSI/ISO Fortran 95 Standard.

E.1. VSI Fortran Language Extensions

This section summarizes the VSI Fortran language extensions. Most extensions are available on all supported operating systems. However, some extensions are limited to one or more platforms. If an extension is limited, it is labeled.

Source Forms

The following are extensions to the methods and rules for source forms:

- Tab-formatting as a method to code lines (see Section 2.3.2.2)
- The letter D as a debugging statement indicator in column 1 of fixed or tab source form (see Section 2.3.2)
- An optional statement field width of 132 columns for fixed or tab source form (see Section 2.3.2)
- An optional sequence number field for fixed source form (see Section 2.3.2.1)
- Up to 511 continuation lines in a source program (see Section 2.3)

Names

The following are extensions to the rules for names (see Section 2.1.2):

- Names can contain up to 63 characters
- The dollar sign ($) is a valid character in names, and can be the first character

Character Sets

The following are extensions to the Fortran 95 character set:

- The Tab (**Tab**) character (see Section 2.2)
- The DEC Multinational extension to the ASCII character set (see Section C.2)

Intrinsic Data Types

The following are data-type extensions (see Section 3.2):

BYTE	INTEGER*1	REAL*8

LOGICAL*1	INTEGER*2	REAL*16
LOGICAL*2	INTEGER*4	COMPLEX*8
LOGICAL*4	INTEGER*8	COMPLEX*16
LOGICAL*8	REAL*4	COMPLEX*32

Constants

C strings are allowed in character constants as an extension (see Section 3.2.5.1).

Hollerith constants are allowed as an extension (see Section 3.4.4).

Expressions and Assignment

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary (see Section 4.1.1.2).

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed (see Section 3.4).

The following are extensions allowed in logical expressions (see Section 4.1.4):

- .XOR. as a synonym for .NEQV.
- Integers as valid logical items

Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC and STATIC (see Section 5.3)
- VOLATILE (see Section 5.19)

Execution Control

The following control statements are extensions to Fortran 95 (see Chapter 7):

- ASSIGN
- Assigned GO TO
- PAUSE

These are older Fortran features that have been deleted in Fortran 95. VSI Fortran fully supports these features.

Built-In Functions

The %VAL, %REF, %DESCR, and %LOC built-in functions are extensions (see Section 8.8.9).

I/O Formatting

The following are extensions allowed in I/O formatting:

- The Q edit descriptor (see Section 11.4.9)
- The dollar sign ($) edit descriptor (see Section 11.4.8 and carriage-control character (see Section 11.8)
- The backslash (\) edit descriptor (see Section 11.4.8)
- The ASCII NUL carriage-control character (see Section 11.8)
- Variable format expressions (see Section 11.7)
- The H edit descriptor (see Section 11.5.2)

  This is an older Fortran feature that has been deleted in Fortran 95. VSI Fortran fully supports this feature.

Compilation Control Statements

The following statements are extensions that can influence compilation (see Chapter 13):

- INCLUDE statement format:

  ```
  INCLUDE '[text-lib] (module-name) [/[NO]LIST]'
  ```

- OPTIONS statement:

  /CHECK = { ALL | [NO]BOUNDS | [NO]OVERFLOW | [NO]UNDERFLOW | NONE }

  /NOCHECK

  /CONVERT = { BIG_ENDIAN | CRAY | FDX | FGX | IBM | LITTLE_ENDIAN | NATIVE | VAXD | VAXG }

  /[NO]EXTEND_SOURCE

  /[NO]F77

  /FLOAT = { D_FLOAT | G_FLOAT | IEEE_FLOAT }

  /[NO]G_FLOATING

  /[NO]I4

  /[NO]RECURSIVE

I/O Statements

The following I/O statements and specifiers are extensions:

- REWRITE statement (see Section 10.7)

- ACCEPT statement (see Section 10.4)
- TYPE statement; a synonym for the PRINT statement (see Section 10.6)
- A key-field-value specifier as a control list parameter (see Section 10.2.1.5)
- A key-of-reference specifier as a control list parameter (see Section 10.2.1.6)
- Indexed READ statement (see Section 10.3.3)
- Indexed WRITE statement (see Section 10.5.3)

File Operation Statements

The following statement specifiers and statements are extensions (see Chapter 12 for details):

- CLOSE statement specifiers:
  - STATUS values: 'SAVE' (as a synonym for 'KEEP'), 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT/DELETE'
  - DISPOSE (or DISP)
- DELETE statement
- INQUIRE statement specifiers:
  - ACCESS value: ’KEYED ’
  - BLOCKSIZE
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN'
  - KEYED
  - ORGANIZATION
  - RECORDTYPE
- OPEN statement specifiers:
  - ACCESS values: ’KEYED ’, 'APPEND'
  - ASSOCIATEVARIABLE
  - BLOCKSIZE
  - BUFFERCOUNT

- BUFFERED
- CARRIAGECONTROL
- CONVERT
- DEFAULTFILE
- DISPOSE
- EXTENDSIZE
- FORM value: 'BINARY'
- INITIALSIZE
- KEY
- MAXREC
- NAME as a synonym for FILE
- NOSPANBLOCKS
- ORGANIZATION
- READONLY
- RECORDSIZE as a synonym for RECL
- RECORDTYPE
- SHARED
- TYPE as a synonym for STATUS
- USEROPEN

- UNLOCK statement

Compiler Directives

The following general directives are extensions (see Chapter 14):

ALIAS	INTEGER	PSECT
ATTRIBUTES	IVDEP	REAL
DECLARE	MESSAGE	SUBTITLE
DEFINE	NODECLARE	STRICT
FIXEDFORMLINESIZE	NOFREEFORM	TITLE
FREEFORM	NOSTRICT	UNDEFINE
IDENT	OBJCOMMENT	UNROLL
IF	OPTIONS	

IF DEFINED PACK

Intrinsic Procedures

The following intrinsic procedures are extensions (see Chapter 9):

ACOSD	AIMAX0	AIMIN0	AJMAX0
AJMIN0	AND	ASIND	ASM
ATAN2D	ATAND	BITEST	BJTEST
CDABS	CDCOS	CDEXP	CDLOG
CDSIN	CDSQRT	COSD	COTAN
COTAND	CQABS	CQCOS	CQEXP
CQLOG	CQSIN	CQSQRT	DACOSD
DASIND	DASM	DATAN2	DATAN2D
DATAND	DATE	DBLEQ	DCMPLX
DCONJG	DCOSD	DCOTAN	DCOTAND
DFLOAT	DFLOTI	DFLOTJ	DIMAG
DREAL	DSIND	DTAND	EOF
ERRSNS	EXIT	FASM	FLOATI
FLOATJ	FP_CLASS	FREE	HFIX
IARGCOUNT	IARGPTR	IBCHNG	IDATE
IIABS	IIAND	IIBCLR	IIBITS
IIBSET	IIDIM	IIDINT	IIDNNT
IIEOR	IIFIX	IINT	IIOR
IIQINT	IIQNNT	IISHFT	IISHFTC
IISIGN	IMAX0	IMAX1	IMIN0
IMIN1	IMOD	IMVBITS	ININT
INOT	INT1	INT2	INT4
IQINT	IQNINT	ISHA	ISHC
ISHL	ISNAN	IZEXT	JFIX
JIAND	JIBCLR	JIBITS	JIBSET
JIDIM	JIDINT	JIDNNT	JIEOR
JINT	JIOR	JIQINT	JIQNNT
JISHFT	JISHFTC	JISIGN	JMAX0
JMAX1	JMIN0	JMIN1	JMOD
JMVBITS	JNINT	JNOT	JZEXT
KIQINT	KIQNNT	LEADZ	LOC
LSHIFT	MALLOC	MULT_HIGH	NWORKERS
OR	POPCNT	POPPAR	QABS
QACOS	QACOSD	QASIN	QASIND
QATAN	QATAND	QATAN2	QATAN2D
QCMPLX	QCONJG	QCOS	QCOSD

QCOSH	QCOTAN	QCOTAND	QDIM
QEXP	QEXT	QEXTD	QFLOAT
QIMAG	QINT	QLOG	QLOG10
QMAX1	QMIN1	QMOD	QNINT
QREAL	QSIGN	QSIN	QSIND
QSINH	QSQRT	QTAN	QTAND
QTANH	RAN	RANDU	RSHIFT
SECNDS	SIND	SIZEOF	SNGLQ
TAND	TIME	TRAILZ	XOR
ZABS	ZCOS	ZEXP	ZEXT
ZLOG	ZSIN	ZSQRT	

The argument KIND is an extension available in the following intrinsic procedures (see Chapter 9):

COUNT	LEN_TRIM	SHAPE	ZEXT
INDEX	MAXLOC	SIZE	
LBOUND	MINLOC	UBOUND	
LEN	SCAN	VERIFY	

See Appendix B for additional language extensions that facilitate compatibility with other versions of Fortran.