DIGITAL Visual Fortran

Language Reference

Date:	December, 1998
Software Version:	DIGITAL Visual Fortran Version 6.0, Standard and Professional Editions
Operating Systems:	Microsoft® Windows® 95, Windows 98, or Windows NT® Version 4

Digital Equipment Corporation
Maynard, Massachusetts

Copyright Page

Introduction

This manual contains the description of the DIGITAL Visual Fortran programming language. It contains information on language syntax and semantics, on adherence to various Fortran standards, and on extensions to those standards.

This manual is intended for experienced applications programmers who have a basic understanding of Fortran concepts and the Fortran 90 language.

Some familiarity with your operating system is helpful. This manual is not a Fortran or programming tutorial.

This document contains the following major sections (this color denotes a link):

Section	Description
Overview	Describes language standards, language compatibility, and Fortran 90 features.
Program Structure, Characters, and Source Forms	Describes program structure, the Fortran 90 character set, and source forms.
Data Types, Constants, and Variables	Describes intrinsic and derived data types, constants, variables (scalars and arrays), and substrings.
Expressions and Assignment Statements	Describes expressions and assignment.
Specification Statements	Describes specification statements, which declare the attributes of data objects.
Dynamic Allocation	Describes dynamic allocation of data objects.
Execution Control	Describes constructs and statements that can transfer control within a program.
Program Units and Procedures	Describes program units (including modules), subroutines and functions, and procedure interfaces.
Intrinsic Procedures	Contains general information on Visual Fortran intrinsic procedures. Each intrinsic is fully described in the A-Z Reference.
Data Transfer I/O Statements	Describes data transfer input/output (I/O) statements.
I/O Formatting	Describes the rules for I/O formatting.
File Operation I/O Statements	Describes auxiliary I/O statements you can use to perform file operations on Windows NT and Windows 95 systems.

Compilation Control Statements and Compiler Directives	Describes compilation control statements and compiler directives.
Scope and Association	Describes scope and association.
Obsolescent and Deleted Language Features	Describes obsolescent language features in Fortran 90 and Fortran 95.
Additional Language Features	Describes some statements and language features supported for programs written in older versions of Fortran.
Character and Key Code Charts	Describes the Visual Fortran character sets available on Windows NT and Windows 95 systems.
Data Representation Models	Describes data representation models for numeric intrinsic functions.
FORTRAN 77 Syntax	Summarizes the syntax for features of the ANSI FORTRAN 77 Standard.
Summary of Language Extensions	Summarizes DIGITAL Fortran extensions to the Fortran 90 Standard.
A-Z Reference	Organizes the functions, subroutines, and statements available in Visual Fortran by the operations they perform. Also has descriptions of all Visual Fortran statements and intrinsics (arranged in alphabetical order).
Glossary	Contains abbreviated definitions of some commonly used terms in this manual.

Other Sources of Information

This section alphabetically lists some commercially published documents that provide reference or tutorial information on Fortran 90 and Fortran 95:

- *Fortran 90 Explained* by M. Metcalf and J. Reid; published by Oxford University Press, ISBN 0-19-853772-7.
- *Fortran 90/95 Explained* by M. Metcalf and J. Reid; published by Oxford University Press, ISBN 0-19-851888-9.
- *Fortran 90/95 for Scientists and Engineers* by S. Chapman; published by McGraw-Hill, ISBN 0-07-011938-4.
- *Fortran 90 Handbook* by J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener; published by Intertext Publications (McGraw-Hill), ISBN 0-07-000406-4.
- *Fortran 90 Programming* by T. Ellis, I. Philips, and T. Lahey; published by Addison-Wesley, ISBN 0201-54446-6.

- *Introduction to Fortran 90/95* by Stephen J. Chapman; published by McGraw-Hill, ISBN 0-07-011969-4.

- *Programmer's Guide to Fortran 90, Second Edition* by W. Brainerd, C. Goldberg, and J. Adams; published by Unicomp, ISBN 0-07-000248-7.

DIGITAL does not endorse these books or recommend them over other books on the same subjects.

Language Reference Conventions

This section discusses the following:

General Conventions

The *Language Reference* uses the following general conventions. (Note that in most cases, blanks are not significant in Fortran 90.) Text in this color denotes a link.

When you see this	Here is what it means
Extensions to Fortran 90	Dark teal type indicates extensions to the Fortran 90 Standard. These extensions may or may not be implemented by other compilers that conform to the language standard.
OUT.TXT, ANOVA.EXE, COPY, LINK, FL32	Uppercase (capital) letters indicate filenames and MS-DOS®-level commands used in the command console. Uppercase is also used for command-line options (unless the application accepts only lowercase).

`!  Comment line`
`WRITE (*,*) 'Hello &`
`&World'`
	This kind of type is used for program examples, program output, and error messages within the text. An exclamation point marks the beginning of a comment in sample programs. Continuation lines are indicated by an ampersand (&) after the code at the end of a line to be continued and before the code on the following line.

AUTOMATIC, INTRINSIC, WRITE	Bold capital letters indicate Fortran 90 statements, functions, subroutines, and keywords. Keywords are a required part of statement syntax, unless enclosed in brackets as explained below. In the sentence, "The following steps occur when a **DO WHILE** statement is executed," the phrase **DO WHILE** is a Fortran 90 keyword.
other keywords	Bold lowercase letters are used for keywords of other languages. In the sentence, "A Fortran 90 subroutine is the equivalent of a function of type **void** in C," the word **void** is a keyword of C.
expression	Words in italics indicate placeholders for information that you must supply. A file-name is an example of this kind of information. Italics are also used to introduce new terms.
[optional item]	Items inside single square brackets are optional. In some examples, square brackets are used to show arrays.

{*choice1* | *choice2*}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in square brackets.
`s[, s]...`	A horizontal ellipsis (three dots) following an item indicates that the item preceding the ellipsis can be repeated. In code examples, a horizontal ellipsis means that not all of the statements are shown.
KEY NAMES	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key. The carriage-return key, sometimes marked with a bent arrow, is referred to as ENTER. The cursor arrow keys on the numeric keypad are called DIRECTION keys. Individual DIRECTION keys are referred to by the direction of the arrow on the key top (LEFT ARROW, RIGHT ARROW, UP ARROW, DOWN ARROW) or the name on the key top (PAGE UP, PAGE DOWN). The key names used in this manual correspond to the names on the IBM® Personal Computer keys. Other machines may use different names.
Compatibility line	The projects or libraries listed are compatible with the language element described. (See the A-Z Summary.)
Fortran	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 90, and DIGITAL Fortran.
Fortran 90	This term refers to language information that is common to ANSI/ISO Fortran 90 and DIGITAL Fortran.
Fortran 95	This term refers to language information that is common to ANSI/ISO Fortran 95 and DIGITAL Fortran.
integer	This term refers to the INTEGER(KIND=1), INTEGER(KIND=2), INTEGER (INTEGER(KIND=4)), and INTEGER(KIND=8) data types as a group.
real	This term refers to the REAL (REAL(KIND=4)), DOUBLE PRECISION (REAL(KIND=8)), and REAL(KIND=16) data types as a group.
complex	This term refers to the COMPLEX (COMPLEX(KIND=4)) and DOUBLE COMPLEX (COMPLEX(KIND=8)) data types as a group.
logical	This term refers to the LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL (LOGICAL(KIND=4)), and LOGICAL(KIND=8) data types as a group.

Syntax Conventions

The *Language Reference* uses certain conventions for language syntax. For example, consider the following syntax for the PARAMETER statement:

PARAMETER [(] *c* = *expr* [, *c* = *expr*]...[)]

This syntax shows that to use this statement, you must specify the following:

- The keyword **PARAMETER**
- An optional left parenthesis
- One or more *c=expr* items, where *c* is a named constant and *expr* is a value; note that a comma must appear between c=expr items.
  The three dots following the syntax means you can enter as many *c=expr* items as you like.
- An optional terminating right parenthesis

The dark teal brackets ([ ]) indicate that the optional parentheses are an extension to standard Fortran.

Platform Labels

A platform is a combination of operating system and central processing unit (CPU) that provides a distinct environment in which to use a product (in this case, a language). For example, Microsoft® Windows® 95 on Intel® is a platform.

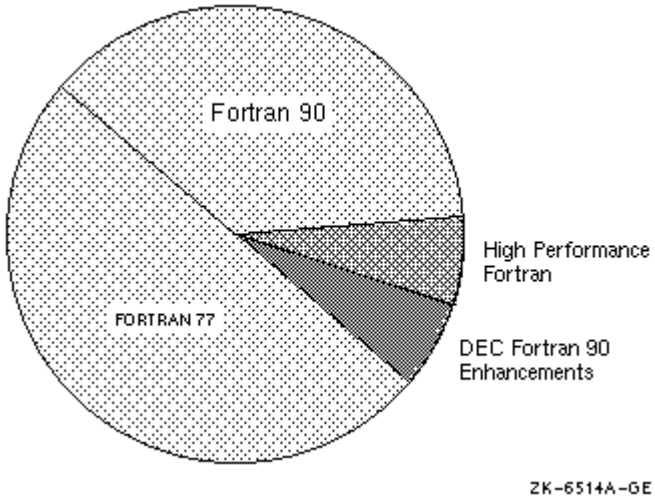
Information applies to all supported platforms unless it is otherwise labeled for a specific platform (or platforms), as follows:

VMS	Applies to OpenVMS™ on Alpha processors.
U*X	Applies to DIGITAL UNIX® on Alpha processors.
WNT	Applies to Microsoft Windows NT® on Alpha and Intel processors.
W95	Applies to Microsoft Windows 95 on Intel processors.
Alpha	Applies to OpenVMS, DIGITAL UNIX, and Microsoft Windows NT on Alpha processors. Only the Professional Edition of Visual Fortran supports Alpha processors (see System Requirements and Optional Software in *Getting Started*).
x86	Applies to Microsoft Windows NT and Windows 95 on Intel processors (see System Requirements and Optional Software in *Getting Started*).

Overview

This chapter discusses DIGITAL Fortran standards conformance and language compatibility, and provides an overview of Fortran 90, Fortran 95, and High Performance Fortran features.

Graphic Representation of DIGITAL Fortran



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

For more information, see:

- Language Standards Conformance
- Language Compatibility
- Fortran 90 Features
- Fortran 95 Features

Language Standards Conformance

DIGITAL Fortran conforms to American National Standard Fortran 90 (ANSI X3.198-1992)[1], American National Standard Fortran 95 (ANSI X3J3/96-007)[2], and includes support for the High Performance Fortran Language Specification.

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

DIGITAL Fortran provides a number of extensions to the Fortran 90 Standard. In the language reference manual, extensions are displayed in this color.

DIGITAL 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.

---

[1]This is the same as International Standards Organization standard ISO/IEC 1539:1991 (E).
[2]This is the same as International Standards Organization standard ISO/IEC 1539-1:1996.

Language Compatibility

DIGITAL Fortran is highly compatible with DIGITAL Fortran 77 (on DIGITAL UNIX Alpha systems, OpenVMS Alpha systems, Windows NT Alpha systems, OpenVMS VAX systems, and ULTRIX RISC systems). It is substantially compatible with PDP-11 FORTRAN 77.

Fortran 90 Features

Fortran 90 offers significant enhancements to FORTRAN 77. Some of the features of Fortran 90 were implemented in earlier versions of DIGITAL Fortran. This topic defines terms and concepts of Fortran 90 and provides an overview of new features.

Certain features of FORTRAN 77 have been replaced by more efficient statements and routines in Fortran 90. These features are listed in Obsolescent Features.

Each section includes tables that list statements, procedures, parameters, attributes, and other features new to Fortran 90. Italicized terms are defined in the Glossary. Each topic includes a cross-reference to other topics that contain more detailed information.

Although most FORTRAN 77 functionality remains unchanged in Fortran 90, some features need special handling. For more information on building Fortran programs, refer to Compatibility Information.

For more information, see:

- New Features
- Improved Features

New Features

The following Fortran 90 features are new to Fortran:

- Free source form

  Fortran 90 provides a new 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 Free Source Form.

- Modules

  Fortran 90 provides a new 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 MODULE PROCEDURE.

- User-defined (derived) data types and operators

  Fortran 90 lets you define new 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 Derived Data Types and Defining Generic Operators.

- Array operations and features

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

  New 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).

  New 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 Arrays and Intrinsic Procedures.

- 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 Defining Generic Names for Procedures.

- 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 POINTER -- Fortran 90 and Pointer Assignments.

- Recursion

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

  For more information, see Program Units and Procedures.

- 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 Procedure Interfaces.

- 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 your user manual or programmer's guide. See alsoObsolescent and Deleted Language Features.

Improved Features

The following Fortran 90 features improve previous Fortran features:

- 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 Program Structure, Characters, and Source Forms.

- 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 new intrinsic functions that allow you to specify numeric precision and inquire about precision characteristics available on a processor.

  For more information, see Data Types, Constants, and Variables and Intrinsic Procedures.

- 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 Program Units and Procedures.

- 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 Data Transfer I/O Statements and on **OPEN** and **INQUIRE**, see File Operation I/O Statements.

- Additional control constructs

  Fortran 90 provides a new 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 Execution Control.

- Additional intrinsic procedures

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

  For more information, see Program Units and Procedures.

- Additional specification statements

  The following specification statements are new in Fortran 90:

  - The INTENT statement
  - The OPTIONAL statement
  - The Fortran 90 POINTER statement
  - The PUBLIC and PRIVATE statements
  - The TARGET statement

- 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 Type Declaration Statements.

- 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 Scope and Association.

Fortran 95 Features

Fortran 95 includes Fortran 90 and most features of FORTRAN 77.

This section briefly describes the Fortran 95 language features that have been implemented by Visual Fortran:

- The 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.

  DIGITAL Fortran previously provided the **FORALL** statement and construct as language extensions.

- 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.

  DIGITAL Fortran previously provided pure procedures as a language extension.

- 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.

- Pointer initialization

  In Fortran 90, there was no way to define the initial value of a pointer or 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.

- Derived-type structure default initialization

  Fortran 95 lets you specify, in derived-type definitions, default initial values for derived-type components.

- Automatic deallocation of allocatable arrays

  Arrays declared using the ALLOCATABLE attribute can now be automatically deallocated in cases where Fortran 90 would have assigned them undefined allocation status. For more

information, see Deallocation of Allocatable Arrays.

DIGITAL Fortran previously provided this feature.

- CPU_TIME intrinsic subroutine

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

- Enhanced CEILING and FLOOR intrinsic functions

  KIND can now be specified for these intrinsic functions.

- Enhanced MAXLOC and MINLOC intrinsic functions

  DIM can now be specified for these intrinsic functions. DIGITAL Fortran previously provided this feature as a language extension.

- Enhanced SIGN intrinsic function

  If the compiler option /assume:minus0 is specified, the **SIGN** function can now distinguish between positive and negative zero (if the processor is capable of doing so).

- Printing of -0.0

  If the compiler option /assume:minus0 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.

- Generic identifier allowed in END INTERFACE statement

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

- 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. DIGITAL Fortran previously provided this feature as a language extension.

- New obsolescent features

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

DIGITAL Fortran **fully supports** features deleted in Fortran 95.

Program Structure, Characters, and Source Forms

This section contains information on the following topics:

- An overview of program structure, including general information on statements and names
- Character sets
- Source forms

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 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.

The following sections discuss Statements, Names, and Keywords.

Statements

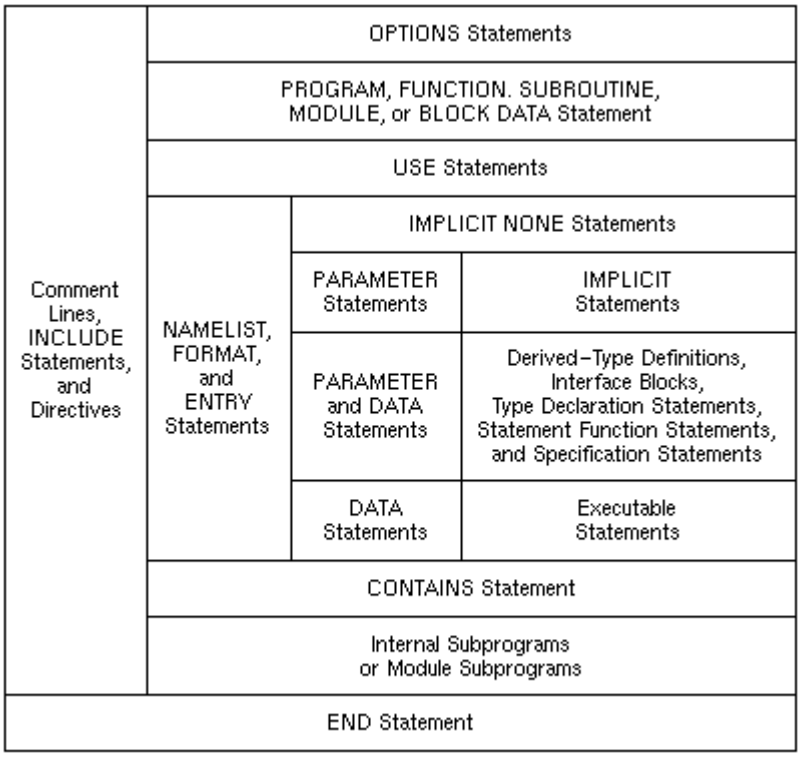
Program statements are grouped into two general classes: executable and nonexecutable. An *executable statement* specifies an action to be performed. A *nonexecutable statement* describes program attributes, such as the arrangement and characteristics of data, as well as editing and data-conversion information.

Order of Statements in a Program Unit

The following figure 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.

Required Order of Statements



PUBLIC and **PRIVATE** statements are only allowed in the scoping units of modules. The following table shows other statements restricted from different types of scoping units.

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.

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, an underscore (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign..

In Fortran 90, a name can contain up to 31 characters. DIGITAL 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.

Note: Be careful when defining names that contain dollar signs.

On OpenVMS systems, naming conventions reserve names containing dollar signs to those created by DIGITAL. On DIGITAL UNIX, Windows NT, and Windows 95 systems, a dollar sign can be a symbol for command or symbol substitution in various shell and utility commands.

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.

The following are all valid examples of using names:

```
INTEGER (SHORT) K       !K names an integer variable
SUBROUTINE EXAMPLE      !EXAMPLE names the subroutine
LABEL: DO I = 1,N       !LABEL names the DO block
```

Keywords

A keyword can either be a part of the syntax of a statement (statement keyword), or it can be the name of a dummy argument (argument keyword). Examples of statement keywords are **WRITE**, **INTEGER**, **DO**, and **OPEN**. Examples of argument keywords are arguments to the intrinsic functions.

In the intrinsic function **UNPACK** (VECTOR, MASK, FIELD), for example, VECTOR, MASK, and FIELD are argument keywords. They are dummy argument names, and any variable may be substituted in their place. Dummy argument names and real argument names are discussed in Program Units and Procedures.

Keywords are not reserved. The compiler recognizes keywords by their context. For example, a program can have an array named IF, read, or Goto, even though this is not good programming practice. The only exception is the keyword **PARAMETER**. If you plan to use variable names beginning with PARAMETER in an assignment statement, you need to use the compiler option /altparam.

Using keyword names for variables makes programs harder to read and understand. For readability, and to reduce the possibility of hard-to-find bugs, avoid using names that look like parts of Fortran statements. Rules that describe the context in which a keyword is recognized are discussed in Program Units and Procedures.

Argument keywords are a feature of Fortran 90 that let you specify dummy argument names when calling intrinsic procedures, or anywhere an interface (either implicit or explicit) is defined. Using argument keywords can make a program more readable and easy to follow. This is described more fully in Program Units and Procedures. The syntax statements in the *A-Z Reference* show the dummy keywords you can use for each Fortran procedure.

Character Sets

DIGITAL Fortran supports the following characters:

- The Fortran 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
blank 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 certain special character sets.

  Printable characters that are not in the Fortran 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 detailed information on character sets and default character types, see Data Types, Constants, and Variables and Using National Language Support Routines. For more information on the ASCII character set, see ASCII and Key Code Charts.

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, see comment indicators in free source form, or fixed and tab source forms.

- 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. DIGITAL 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, see continuation indicators in free source form, or fixed and tab source

forms.

The following table summarizes characters used as indicators in source forms:

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
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] Fixed and tab forms only.

Fixed source form is the default for files with a .FOR extension. You can select free source form in one of three ways:

- Use the file extension .F90 for your source file.
- Use the compiler option /free.
- Use the FREEFORM compiler directive in the source file.

Source form and line length can be changed at any time by using the FREEFORM, NOFREEFORM, or FIXEDFORMLINESIZE directives. The change remains in effect until the end of the file, or until changed again.

Source code can be written so that it is useable for all source forms.

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.

Free Source Form

In free source form, statements are not limited to specific positions on a source line, and a line can contain from 0 to 132 characters.

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 Source Forms.

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 90 permits up to 39 continuation lines in free-form programs, DIGITAL 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 indent the continuation line of a character constant, an ampersand must be the first character of the continued line; otherwise, the blanks at the beginning of the continuation line will be included as part of the character constant. For example:

```
ADVERTISER = "Davis, O'Brien, Chalmers & Peter&
                &son"
```

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 details on the general rules for all source forms, see Source Forms.

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 compiler option /extend_source 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, DIGITAL 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 the /pad_source 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 90 permits up to 19 continuation lines in a fixed-form program, DIGITAL 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 the /d_lines option to force the compiler to treat debugging statements as source text to be compiled.

The following sections discuss Fixed-format lines and Tab-format lines.

For details on the general rules for all source forms, see Source Forms.

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 the /extend_source 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 DIGITAL 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.

For details on the general rules for all source forms, see Source Forms.

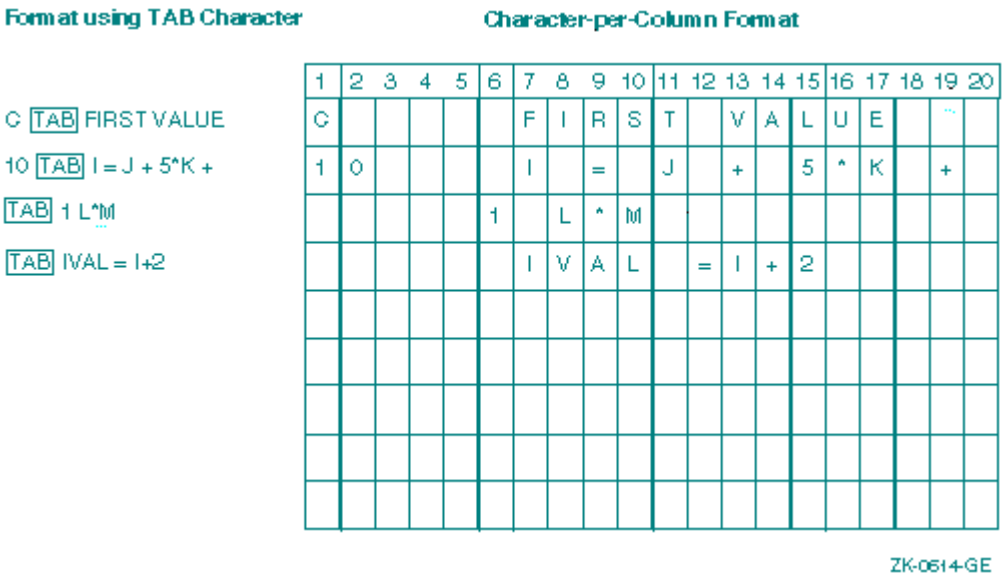
For details on the general rules for fixed and tab source forms, see Fixed and Tab Source Forms.

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.

The following figure shows equivalent source lines coded with tab and fixed source form.

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 DIGITAL 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:** Do not use tabs to position sequence numbers, or the compiler may interpret the sequence numbers as part of the statement fields in your program.

For details on the general rules for all source forms, see Source Forms.

For details on the general rules for fixed and tab source forms, see Fixed and Tab Source Forms.

Source Code Useable for All Forms

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

Blanks	Treat as significant (see Free Source Form).
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
```

Data Types, Constants, and Variables

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:

For More Information:

- See Type Declaration Statements.

- See Defined Operations.
- See the PARAMETER attribute and statement
- On valid operations for data types, see Expressions.
- On ranges for numeric literal constants, see your programmer's guide.
- On named constants, see PARAMETER.

Intrinsic Data Types

DIGITAL Fortran provides the following intrinsic data types:

- INTEGER

  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
    This kind is only available on Alpha processors.

- REAL

  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
    This kind is only available on OpenVMS and DIGITAL UNIX systems.

- DOUBLE PRECISION

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

- COMPLEX

  There are two kind parameters for data of type complex:

  - COMPLEX([KIND=]4) or COMPLEX*8
  - COMPLEX([KIND=]8) or COMPLEX*16

- DOUBLE COMPLEX

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

- LOGICAL

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
  This kind is only available on Alpha processors.

- CHARACTER

  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
  ...
```

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

For More Information:

- See the KIND intrinsic function.
- On declaration statements for intrinsic data types, see Declaration Statements for Noncharacter Types and Declaration Statements for Character Types.
- On operations for intrinsic data types, see Expressions.
- On storage requirements for intrinsic data types, see the Data Type Storage Requirements table.

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. Kind 8 is only available on Alpha processors.

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.

You can change the result of a default specification by using the /integer_size:*size* compiler option or the INTEGER compiler directive.

The intrinsic inquiry function KIND returns the kind type parameter, if you do not know it. You can use the intrinsic function SELECTED_INT_KIND to find the kind values that provide a given range of integer values. The decimal exponent range is returned by the intrinsic function RANGE.

For more information on the integer data type, see Integer Constants.

Examples

The following examples show ways an integer variable can be declared.
An entity-oriented example is:

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

An attribute-oriented example is:

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

An integer can be used in certain cases when a logical value is expected, such as in a logical expression evaluating a condition, as in the following:

```
INTEGER I, X
READ (*,*) I
IF (I) THEN
  X = 1
END IF
```

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.

Integer constants are interpreted as decimal values (base 10) by default. To specify a constant that is not in base 10, use the following extension 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 type for integers.

The following seven integers are all assigned a value equal to 3,994,575 decimal:

```
I = 2#11110011110011111001111
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 Data**	**Hexadecimal Value in 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'

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. Kind 16 is only available on OpenVMS and DIGITAL UNIX systems.

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.

You can change the result of a default specification by using the /real_size:*size* compiler option or the

REAL compiler directive.

The intrinsic inquiry function KIND returns the kind type parameter. The intrinsic inquiry function RANGE returns the decimal exponent range, and the intrinsic function PRECISION returns the decimal precision. You can use the intrinsic function SELECTED_REAL_KIND to find the kind values that provide a given precision and exponent range.

For more information on real data types, see General Rules for Real Constants, REAL(4) Constants, and REAL(8) or DOUBLE PRECISION Constants.

Examples

The following examples show how real variables can be declared.
An entity-oriented example is:

```
REAL (KIND = high), OPTIONAL :: testval
REAL, SAVE :: a(10), b(20,30)
```

An attribute-oriented example is:

```
REAL (KIND = high) testval
REAL a(10), b(20,30)
OPTIONAL testval
SAVE a, b
```

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) (VMS, U*X).
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 is (by default) a single-precision (REAL(4)) constant. You can change the default behavior by specifying the compiler option /fpconstant.

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.

On OpenVMS and DIGITAL UNIX systems, 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.

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.

On DIGITAL UNIX, Windows NT, and Windows 95 systems, IEEE® S_floating format is used. On OpenVMS systems, either DIGITAL VAX F_floating or IEEE S_floating format is used, depending on the compiler option specified.

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:

- See General Rules for Real Constants.
- On the format and range of REAL(4) data, see your programmer's guide.
- On compiler options affecting REAL data, see your programmer's guide.

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.

On DIGITAL UNIX, Windows NT, and Windows 95 systems, IEEE T_floating format is used. On OpenVMS systems, either DIGITAL VAX D_floating, G_floating, or IEEE T_floating format is used, depending on the compiler option specified.

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:

- See General Rules for Real Constants.
- On the format and range of DOUBLE PRECISION (REAL(8)) data, see your programmer's guide.
- On compiler options affecting DOUBLE PRECISION (REAL(8)) data, see your programmer's guide.

REAL(16) Constants (VMS, U*X)

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:

- See General Rules for Real Constants.
- On the format and range of REAL(16) data, see the *DIGITAL Fortran User's Guide*.

Complex Data Types

Complex data types can be specified as follows:

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

n
Is kind 4 or 8.

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

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.

For more information on complex data types, see General Rules for Complex Constants, COMPLEX (4) Constants, and COMPLEX(8) or DOUBLE COMPLEX Constants.

Examples

The following examples show how complex variables can be declared.
An entity-oriented example is:

```
COMPLEX (4), DIMENSION (8) :: cz, cq
```

An attribute-oriented example is:

```
COMPLEX(4) cz, cq
DIMENSION(8) cz, cq
```

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

Note that the comma and parentheses are required.

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 General Rules for Complex Constants and REAL(4) Constants for the rules on forming **REAL**(4) constants.)

The **REAL**(4) constants in a **COMPLEX** constant have one of the following formats:

- On DIGITAL UNIX, Windows NT, and Windows 95 systems: IEEE S_floating format
- On OpenVMS systems: DIGITAL VAX F_floating or IEEE S_ floating format (depending on the compiler option specified)

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:

- See General Rules for Complex Constants.
- On the format and range of COMPLEX (COMPLEX(4)) data, see your programmer's guide.
- On compiler options affecting REAL data, see your programmer's guide.

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 General Rules for Complex Constants and REAL(8) or DOUBLE PRECISION Constants for the rules on forming **DOUBLE PRECISION** constants.)

The **DOUBLE PRECISION** constants in a **COMPLEX**(8) or **DOUBLE COMPLEX** constant have one of the following formats:

- On DIGITAL UNIX, Windows NT, and Windows 95 systems: IEEE T_floating format
- On OpenVMS systems: DIGITAL VAX D_floating, G_floating, or IEEE T_floating format (depending on the compiler option specified)

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:

- See General Rules for Complex Constants.
- On the format and range of DOUBLE COMPLEX data, see your programmer's guide.
- On compiler options affecting DOUBLE COMPLEX data, see your programmer's guide.

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. Kind 8 is only available on Alpha processors.

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.

For more information on logical data types, see Logical Constants.

Examples

The following examples show how logical variables can be declared.
An entity-oriented example is:

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (KIND = byte), SAVE :: doit, dont
```

An attribute-oriented example is:

```
LOGICAL flag1, flag2
LOGICAL (KIND = byte) doit, dont
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

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:

For details on integer data type ranges, see your programmer's guide.

Character Data Type

The character data type can be specified as follows:

CHARACTER
CHARACTER([**KIND**=]*n*)
CHARACTER***len*

n
Is kind 1.

len
Is a string length (not a kind). For more information, see Declaration Statements for Character Types.

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

Several Multi-Byte Character Set (MBCS) functions are available to manipulate special non-English characters. These are described in Using National Language Support Routines.

For more information on the character data type, see Character Constants, C Strings, and Character Substrings.

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*[*ch*...]' [*C*]
[*k*_]"*ch*[*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 C Strings in Character Constants.

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	Explanation
’HEADINGS	No trailing apostrophe.
’Map Number:"	Beginning delimiter does not match ending delimiter.

For More Information, see Declaration Statements for Character Types.

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).

The following table shows the escape sequences that are allowed in character constants:

Table: C-Style Escape Sequences	
Escape Sequence	**Represents**
\a	A bell
\b	A backspace
\f	A formfeed
\n	A new line
\r	A carriage return
\t	A horizontal tab
\v	A vertical tab
\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 string. 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.

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
```

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

For More Information:

- See Arrays.
- See Array Elements.
- See Structure Components.

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.

The following is also discussed in this section:

- Derived-Type Definition
- Default Initialization
- Structure Components
- Structure Constructors

For More Information:

- On OPERATOR interfaces, see Defining Generic Operators.
- On ASSIGNMENT interfaces, see Defining Generic Assignment.
- On intrinsic assignment of derived types, see Derived-Type Assignment Statements.
- On record structures, see Records.

Derived-Type Definition

A derived-type definition specifies the name of a user-defined type and the types of its components. For details on creating derived types, see Derived Type in the *A to Z Reference*.

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.

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 Array Elements.
- On references to array sections, see Array Sections.
- On examples of derived types in modules, see Modules and Module Procedures.

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:

See also Pointer Assignments.

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

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

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 128 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.

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 128 bits in octal (43 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.
`07772'`	No apostrophe after the O.
`"0737"`	No O before the first quotation mark.

For More Information:

See also Alternative Syntax for Octal and Hexadecimal Constants.

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 128 bits in hexadecimal (32 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:

See also Alternative Syntax for Octal and Hexadecimal Constants.

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.

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'10011001111111010011'`	REAL(4)	4
`RAFFIA = Z'99AF2'`	REAL(4)	4
`RALPHA = 4HABCD`	REAL(4)	4
`DOUBLE = B'1111111111100110011010'`	REAL(8)	8
`DOUBLE = Z'FFF99A'`	REAL(8)	8

`DOUBLE = 8HABCDEFGH`	REAL(8)	8
`JCOUNT = ICOUNT + B'011101110111'`	INTEGER(2)	2
`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 on Alpha processors. On Intel processors, a length of four bytes is used.
- 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(4)	4
`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.

  The Data Type Storage Requirements table 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.

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.

An object with subobjects, such as an array, can only be defined when all of its subobjects are defined. Conversely, when at least one of its subobjects are undefined, the object itself, such as an array or derived type, is undefined.

This section also discusses the Data Types of Scalar Variables and Arrays.

For More Information:

- See Type Declaration Statements.
- See the DATA statement.
- See Data Type of a Numeric Expressions.
- On storage association of variables, see Storage Association.

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.

See also Specification of Data Type and Implicit Typing Rules.

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:

- See Type declaration statements.
- See Assumed-length character arguments.
- See the IMPLICIT statement.
- On character type declaration statements, see Declaration Statements for Character Types.

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.

Note: You cannot change the implicit type of a name beginning with a dollar sign in an **IMPLICIT** statement.

For More Information:

- See Type declaration statements.
- See the IMPLICIT statement.

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 atype 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.

This section also discusses:

- Whole Arrays
- Array Elements
- Array Sections
- Array Constructors

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) :: D         ! 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:

- See the DIMENSION attribute.
- See Intrinsic data types.
- See Derived data types.
- On array specifications, see Declaration Statements for Arrays.
- On intrinsic functions that perform array operations, see Categories of Intrinsic Functions.
- On using arrays, see Use Arrays Efficiently.

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
```

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).

You can use functions and array elements as subscripts. For example:

```
REAL A(3, 3)
REAL B(3, 3), C(89), R
B(2, 2) = 4.5                          ! Assigns the value 4.5 to element B(2, 2)
R = 7.0
C(INT(R)*2 + 1) = 2.0                  ! Element 15 of C = 2.0
A(1,2) = B(INT(C(15)), INT(SQRT(R)))  ! Element A(1,2) = element B(2,2) = 4.5
```

For information on forms for array specifications, see Declaration Statements for Arrays.

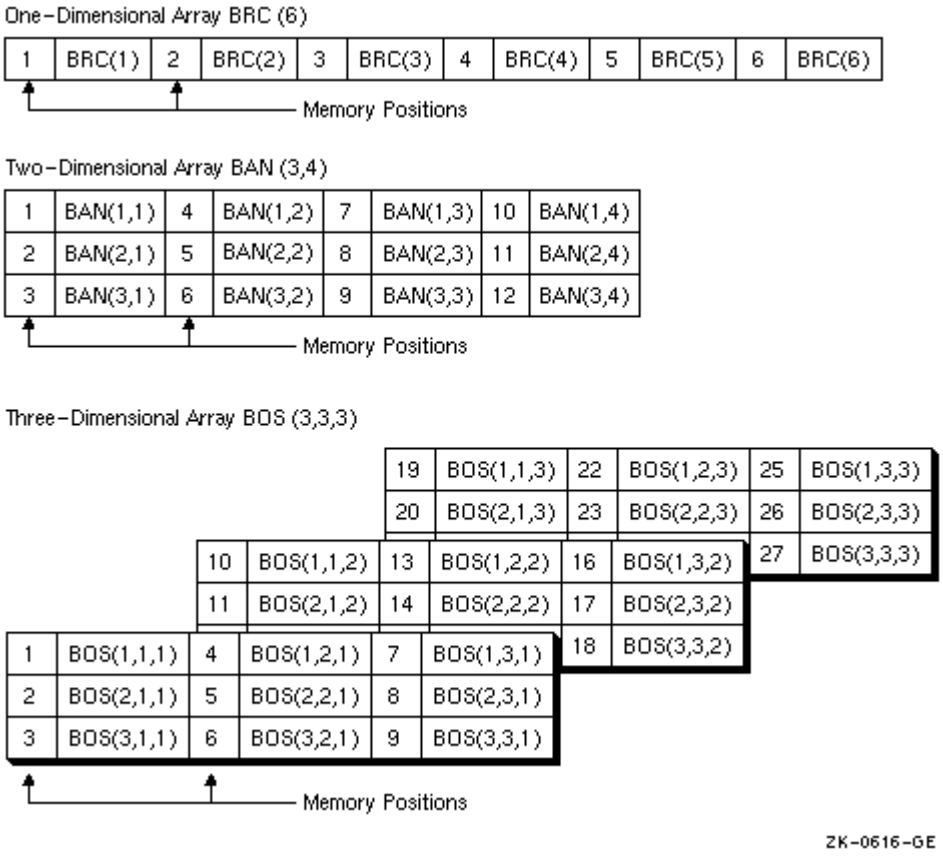
Array Element Order

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.

The following figure shows array storage in one, two, and three dimensions:

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:

- See Array association.
- On substrings, see Character Constants.
- On arrays as structure components, see Structure Components.
- On storage sequence association, see Storage Association.

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:

```
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).)
```

For More Information:

- See the INTENT attribute.
- See the PARAMETER attribute.
- See the TARGET attribute.
- See array sections as Structure components.
- See Array constructors.
- On substrings, see Character Substrings.

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.

If you leave out all subscripts, the section defaults to the entire extent in that dimension. For example:

```
REAL A(10)
A(1:5:2) = 3.0   ! Sets elements A(1), A(3), A(5) to 3.0
A(:5:2) = 3.0    ! Same as the previous statement
                 !   because the lower bound defaults to 1
A(2::3) = 3.0    ! Sets elements A(2), A(5), A(8) to 3.0
                 ! The upper bound defaults to 10
A(7:9) = 3.0     ! Sets elements A(7), A(8), A(9) to 3.0
                 ! The stride defaults to 1
A(:) = 3.0       ! Same as A = 3.0; sets all elements of
                 !   A to 3.0
```

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/)
C = (/2,1,1/)         ! Will result in 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.* For example:

```
REAL A(3, 3), B(4)
INTEGER K(4)
! Vector K has repeated values
K = (/3, 1, 1, 2/)
! Sets all elements of A to 5.0
A = 5.0
B = A(3, K)
```

The array section A(3,K) consists of the elements:

```
A(3, 3) A(3, 1) A(3, 1) A(3, 2)
```

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.

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-list, do-variable = expr1, expr2 [,expr3])

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, each expression must have the same character length.

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.

Implied-**DO** expressions and values can be mixed in the value list of an array constructor. For example:

```
 INTEGER A(10)
 A = (/1, 0, (I, I = -1, -6, -1), -7, -8 /)
 !Mixed values and implied-DO in value list.
```

This example sets the elements of A to the values, in order, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8.

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
```

The following shows another example:

```
INTEGER B(2,3), C(8)
! Assign values to a (2,3) array.
B = RESHAPE((/1, 2, 3, 4, 5, 6/),(/2,3/))
! Convert B to a vector before assigning values to
! vector C.
C = (/ 0, RESHAPE(B,(/6/)), 7 /)
```

For More Information:

- See the DO construct.
- See Subscript triplets.
- See Derived types.
- See Structure constructors.
- On array element order, see Array Elements.
- On another way to assign values to arrays, see Array Assignment Statements.
- On array specifications, see Declaration Statements for Arrays.

Expressions and Assignment Statements

This chapter contains information on the following topics:

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:

- See Arrays.
- See Derived data types.
- On function subprograms that define operators, see Defining Generic Operators.
- On pointers, see the POINTER statement.

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. (.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 mathematically defined. For example, dividing by

zero or raising a zero-valued base to a zero-valued or negative-valued power is invalid. Raising a negative-valued base to a real power is also invalid.

Numeric expressions are evaluated in an order determined by a precedence associated with each operator, as follows (see also Summary of Operator Precedence):

Operator	Precedence
**	Highest
* 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, DIGITAL 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 DIGITAL 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)
```

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.

```
4 + 3 * 2 - 6/2   = 7
  ^   ^   ^  ^
  2   1   4  3


(4 + 3) * 2 - 6/2   = 11
   ^    ^   ^  ^
   1    2   4  3


(4 + 3 * 2 - 6)/2   = 2
   ^   ^   ^   ^
   2   1   3   4


((4 + 3) * 2 - 6)/2   = 4
    ^    ^   ^   ^
    1    2   3   4
```

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)
```

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.

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)[1]	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(4)	.
INTEGER(8)[1]	.
REAL(4)	.
REAL(8)[2]	.
REAL(16)[3]	.
COMPLEX(4)	.
COMPLEX(8)[4]	Highest

[1] Alpha only
[2] DOUBLE PRECISION
[3] VMS, U*X
[4] 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 data type and a DOUBLE PRECISION data type produces a DOUBLE COMPLEX

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. (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 `0.3333333134651184D0`. 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 COMPLEX and DOUBLE PRECISION operands are performed as DOUBLE COMPLEX 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) (VMS, U*X) representation of the constant were given. For example, the expression `1.0D0 + 0.3333333` is treated as if it is `1.0D0 + 0.3333333000000000D0`.

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 '.

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:

For details on the ranking of data types, see Data Type of Numeric Expressions.

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.`	Logical inequivalence (exclusive OR): the expression is true if either A

	B	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:

For details on the precedence of numeric, relational, and logical operators, see Summary of Operator Precedence.

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 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.

Examples

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 Defining Generic Operators.
- On operator precedence, see Summary of Operator Precedence.

Summary of Operator Precedence

The following table shows the precedence of all intrinsic and defined operators:

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

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.

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.

In an initialization expression, the exponential operator (**) must have a power of type integer.

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).

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:

- See Array constructors.
- See Structure constructors.
- See Intrinsic procedures.

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)
- An array constructor where each element, and bounds and strides of each implied-do are expressions whose primaries are restricted expressions
- 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 transformational functions (each argument must be a restricted expression of type integer or character):

REPEAT	**SELECTED_REAL_KIND**
RESHAPE	**TRANSFER**
SELECTED_INT_KIND	**TRIM**

- A reference to the transformational function **NULL**
- A reference to one of the following inquiry functions:

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

  Each function argument must be one of the following:

  - A restricted 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 a restricted expression

- A reference to a specification function (see below) where each argument is a restricted expression
- 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).

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

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:

- See Array constructors.
- See Structure constructors.
- See Intrinsic procedures.
- See Implicit typing rules.
- See Use and host association.
- See PURE procedures.

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, and element array.

The ASSIGN statement assigns a label to an integer variable. It is discussed elsewhere.

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:

- See Arrays.
- See Pointers.
- See Derived data types.
- On subroutine subprograms that define assignment, see Defining Generic Assignment.

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.

The following table summarizes the data conversion rules for numeric assignment statements. REAL (16) is only available on OpenVMS and DIGITAL UNIX systems.

Conversion Rules for Numeric Assignment Statements

Scalar Memory Reference (V)	**Expression (E)**					
	Integer or Logical	**REAL (KIND=4)**	**REAL (KIND=8)**	**REAL (KIND=16) (VMS, U*X)**	**COMPLEX (KIND=4)**	**COMPLEX (KIND=8)**
Integer or logical	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncated E to integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used	Truncate real part of E to integer and assign to V; imaginary part of E is not used
REAL (KIND=4)	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS portion of E to V; LS portion of E is rounded	Assign MS portion of E to V; LS portion of E is rounded	Assign real part of E to V; imaginary part of E is not used	Assign MS portion of the real part of E to V; LS portion of the real part of E is rounded; imagininary part of E is not used
REAL (KIND=8)	Append fraction (.0) to E and assign to V	Assign E to MS portion of V; LS portion of V is 0	Assign E to V	Assign MS portion of E to V; LS portion of E is rounded	Assign real part of E to MS of V; LS portion of V is 0; imaginary part of E is not used	Assign real part of E to V; imaginary part of E is not used
REAL (KIND=16) (VMS, U*X)	Append fraction (.0) to E and assign to V	Assign E to MS portion of V; LS portion of V is 0	Assign E to MS portion of V; LS portion of V is 0	Assign E to V	Assign real part of E to MS of V; LS portion of V is 0; imaginary part of E is not used	Assign real part of E to MS portion of V; LS portion of real part of V is 0; imaginary part of E is not used
COMPLEX (KIND=4)	Append fraction (.0) to E and assign to real	Assign E to real part of V; imaginary part of V is	Assign MS portion of E to real part of V; LS portion of	Assign MS portion of E to real part of V; LS portion of E	Assign E to V	Assign MS portion of real part of E to real part of V; LS portion of real part

	part of V; imaginary part of V is 0.0	0.0	E is rounded; imaginary part of V is 0.0	is rounded; imaginary part of V is 0.0		of E is rounded. Assign MS portion of imaginary part of V; LS portion of imaginary part of E is rounded
COMPLEX (KIND=8)	Append fraction (.0) to E and assign to V; imaginary part of V is 0.0	Assign E to MS portion of real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part is 0.0	Assign MS portion of E to real part of V; LS portion of E is rounded; imaginary part of V is 0.0	Assign real part of E to MS portion of real part of V; LS portion of real part is 0. Assign imaginary part of E to MS portion of imaginary part of V; LS portions of imaginary part is 0.	Assign E to V

```
MS = Most significant (high order) binary digits
LS = Least significant (low order) binary digits
```

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
```

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. (Hollerith constants are numeric, not character.)

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:

- See Derived types.
- See Pointer assignments.

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:

- See Arrays.
- See Array constructors.
- On masked array assignment, see WHERE.
- On element array assignment, see FORALL.

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:

- See Subroutines.
- See Derived data types.
- On subroutine subprograms that define assignment, see Defining Generic Assignment.
- On intrinsic operations, see Numeric Expressions and Character Expressions.

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
<mark>
POINTER_C => NULL ()           ! reference to NULL intrinsic
<endmark>
```

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).

You can use the intrinsic function ASSOCIATED to find out if a pointer is associated with a target or if two pointers are associated with the same target. For example:

```
REAL C (:), D(:), E(5)
POINTER C, D
TARGET E
LOGICAL STATUS
! Pointer assignment.
C => E
! Pointer assignment.
D => E
! Returns TRUE; C is associated.
STATUS = ASSOCIATED (C)
! Returns TRUE; C is associated with E.
STATUS = ASSOCIATED (C, E)
! Returns TRUE; C and D are associated with the
! same target.
STATUS = ASSOCIATED (C, D)
```

For More Information:

- See Arrays.
- See ALLOCATE.
- See ASSOCIATED.
- See DEALLOCATE.
- See NULLIFY.
- See NULL.
- See POINTER.
- See Defined assignments.
- On derived-type intrinsic assignments, see Intrinsic Assignments.

WHERE Statement and Construct

You can perform an array operation on selected elements by using masked array assignment. For more information, see WHERE.

See also FORALL.

FORALL Statement and Construct

The FORALL statement and construct is a generalization of the Fortran 90 masked array assignment. It allows more general array shapes to be assigned, especially in construct form. For more information, see FORALL.

See also WHERE.

Specification Statements

A *specification statement* is a nonexecutable statement that declares the attributes of data objects. In Fortran 90, many of the attributes that can be defined in specification statements can also be optionally specified in type declaration statements.

The following are specification statements:

- Type declaration statement

  Explicitly specifies the properties (for example: data type, rank, and extent) of data objects.

- ALLOCATABLE attribute and statement

  Specifies a list of array names that are allocatable (have a deferred-shape).

- AUTOMATIC and STATIC attributes and statements

  Control the storage allocation of variables in subprograms.

- COMMON statement

  Defines one or more contiguous areas, or blocks, of physical storage (called common blocks).

- DATA statement

  Assigns initial values to variables before program execution.

- DIMENSION attribute and statement

  Specifies that an object is an array, and defines the shape of the array.

- EQUIVALENCE statement

  Specifies that a storage area is shared by two or more objects in a program unit.

- EXTERNAL attribute and statement

  Allows external (user-supplied) procedures to be used as arguments to other subprograms.

- IMPLICIT statement

  Overrides the implicit data type of names.

- INTENT attribute and statement

  Specifies the intended use of a dummy argument.

- INTRINSIC attribute and statement

  Allows intrinsic procedures to be used as arguments to subprograms.

- NAMELIST statement

  Associates a name with a list of variables. This group name can be referenced in some input/output operations.

- OPTIONAL attribute and statement

  Allows a procedure reference to omit arguments.

- PARAMETER attribute and statement

  Defines a named constant.

- POINTER attribute and statement

  Specifies that an object is a pointer.

- PUBLIC and PRIVATE attributes and statements

  Declare the accessibility of entities in a module.

- SAVE attribute and statement

  Causes the definition and status of objects to be retained after the subprogram in which they are declared completes execution.

- TARGET attribute and statement

  Specifies a pointer target.

- VOLATILE attribute and statement

  Prevents optimizations from being performed on specified objects.

For More Information:

- See BLOCK DATA.
- See PROGRAM.

Type Declaration Statements

A type declaration statement explicitly specifies the properties of data objects or functions. For more information, see Type Declarations in the *A to Z Reference*.

This section also discusses:

- Declaration Statements for Noncharacter Types
- Declaration Statements for Character Types
- Declaration Statements for Derived Types
- Declaration Statements for Arrays

For More Information:

- See Derived data types.
- See the DATA statement.
- See Initialization expressions.
- On specific kind parameters of intrinsic data types, see Intrinsic Data Types.
- On implicit typing, see Implicit Typing Rules.
- On explicit typing, see Specification of Data Type.

Declaration Statements for Noncharacter Types

The following table shows the data types that can appear in noncharacter type declaration statements.

Noncharacter Data Types

BYTE[1]
LOGICAL[2]
LOGICAL([KIND=]1) (or LOGICAL*1)
LOGICAL([KIND=]2) (or LOGICAL*2)
LOGICAL([KIND=]4) (or LOGICAL*4)
LOGICAL([KIND=]8) (or LOGICAL*8)[3]
INTEGER[4]
INTEGER([KIND=]1) (or INTEGER*1)
INTEGER([KIND=]2) (or INTEGER*2)
INTEGER([KIND=]4) (or INTEGER*4)
INTEGER([KIND=]8) (or INTEGER*8)[3]
REAL[5]
REAL([KIND=]4) (or REAL*4)
DOUBLE PRECISION (REAL([KIND=]8) or REAL*8)
REAL([KIND=]16) (or REAL*16)[6]
COMPLEX[7]
COMPLEX([KIND=]4) (or COMPLEX*8)
DOUBLE COMPLEX (COMPLEX([KIND=]8) or COMPLEX*16)
1 Same as INTEGER(1).

2 This is treated as default logical.
3 Alpha only.
4 This is treated as default integer.
5 This is treated as default real.
6 VMS, U*X.
7 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*. 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(KIND=2) I, J, K, M12*4, Q, IVEC*4(10)
REAL(KIND=8) WX1, WXZ, WX3*4, WX5, WX6*4
REAL(KIND=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 your programmer's guide.
- On the general form and rules for type declaration statements, see Type Declarations.

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:

Keyword Forms

CHARACTER [([**LEN=**]*len*)]
CHARACTER [([**LEN=**]*len* [, [**KIND=**]*k*])]
CHARACTER [(**KIND=***k* [, **LEN=***len*])]

Nonkeyword Form

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 2147483647 (2**31-1) for DIGITAL UNIX, Windows NT, and Windows 95 systems; 65535 for OpenVMS systems. Negative values are treated as zero.

k
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

In the following example, the character string last_name is given a length of 20:

```
CHARACTER (LEN=20) last_name
```

In the following example, stri is given a length of 12, while the other two variables retain a length of 8.

```
CHARACTER *8 strg, strh, stri*12
```

In the following example, as a dummy argument strh is given the length of an assigned string when it is assigned, while the other two variables retain a length of 8:

```
CHARACTER *8 strg, strh(*), stri
```

The following examples show ways to specify strings of known length:

```
CHARACTER*32 string
CHARACTER string*32
```

The following examples show ways to specify strings of unknown length:

```
CHARACTER string*(*)
CHARACTER*(*) string
```

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.

```
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:

- See Obsolescent features in Fortran 95.
- On asterisk length specifications, see Data Types of Scalar Variables and Assumed-Length Character Arguments.
- On the general form and rules for type declaration statements, see Type Declarations.

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:

- See the TYPE statement.
- See Use and host association.
- See the PUBLIC and PRIVATE attributes.
- See Structure constructors.

- On the general form and rules for type declaration statements, see Type Declarations.

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
- Assumed-shape
- Assumed-size
- Deferred-shape

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:

For details on the general form and rules for type declaration statements, see Type Declarations.

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
```

Consider the following:

```
 INTEGER M(10, 10, 10)
 INTEGER K(-3:6, 4:13, 0:9)
```

M and K are both explicit-shape arrays with a rank of 3, a size of 1000, and the same shape (10,10,10). Array M uses the default lower bound of 1 for each of its dimensions. So, when it is declared only the upper bound needs to be specified. Each of the dimensions of array K has a lower bound other than the default, and the lower bounds as well as the upper bounds are declared.

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
```

Consider the following:

```
 SUBROUTINE EXAMPLE (N, R1, R2)
   DIMENSION A (N, 5), B(10*N)
   ...
   N = IFIX(R1) + IFIX(R2)
```

When the subroutine is called, the arrays A and B are dimensioned on entry into the subroutine with the value of the passed variable N. Later changes to the value of N have no effect on the dimensions of array A or B.

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:

See also Specification expressions.

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.

Examples

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.

Consider the following:

```
SUBROUTINE ASSUMED(A)
  REAL A(:, :, :)
```

The array A has rank 3, indicated by the three colons (:) separated by commas (,). However, the extent of each dimension is unspecified. When the subroutine is called, A takes its shape from the array passed to it. For example, consider the following:

```
REAL X (4, 7, 9)
...
CALL ASSUMED(X)
```

This gives A the dimensions (4, 7, 9). The actual array and the assumed-shape array must have the same rank.

Consider the following:

```
SUBROUTINE ASSUMED(A)
  REAL A(3:, 0:, -2:)
...
```

If the subroutine is called with the same actual array X(4, 7, 9), as in the previous example, the lower and upper bounds of A would be:

```
A(3:6, 0:6, -2:6)
```

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.

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

Examples

The following is an example of an assumed-size specification:

```
SUBROUTINE SUB(A, N)
  REAL A, N
  DIMENSION A(1:N, *)
  ...
```

The following example shows that you can specify lower bounds for any of the dimensions of an assumed-size array, including the last:

```
SUBROUTINE ASSUME(A)
  REAL A(-4:-2, 4:6, 3:*)
```

For More Information:

For details on array element order, see Array Elements.

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.

Examples

The following are examples of deferred-shape specifications:

```
REAL, ALLOCATABLE :: A(:,:)       ! Allocatable array
REAL, POINTER :: C(:), D (:,:,:)  ! Array pointers
```

If a deferred-shape array is declared in a **DIMENSION** or **TARGET** statement, it must be given the ALLOCATABLE or POINTER attribute in another statement. For example:

```
DIMENSION P(:, :, :)
POINTER P

TARGET B(:,:)
ALLOCATABLE B
```

If the deferred-shape array is an array of pointers, its size, shape, and bounds are set in an **ALLOCATE** statement or in the pointer assignment statement when the pointer is associated with an allocated target. A pointer and its target must have the same rank.

For example:

```
REAL, POINTER :: A(:,:), B(:), C(:,:)
INTEGER, ALLOCATABLE :: I(:)
REAL, ALLOCATABLE, TARGET :: D(:, :), E(:)
      ...
ALLOCATE (A(2, 3), I(5), D(SIZE(I), 12), E(98) )
C => D              ! Pointer assignment statement
B => E(25:56)       ! Pointer assignment to a section
                    !   of a target
```

For More Information:

- See the POINTER attribute.
- See the ALLOCATABLE attribute.
- See the ALLOCATE statement.
- See Pointer assignment.
- See the LBOUND intrinsic function.
- See the UBOUND intrinsic function.

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. For more information, see ALLOCATABLE in the *A to Z Reference*.

AUTOMATIC and STATIC Attributes and Statements

The AUTOMATIC and STATIC attributes control the storage allocation of variables in subprograms. For more information, see AUTOMATIC and STATIC in the *A to Z Reference*.

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. For more information, see COMMON in the *A to Z Reference*.

DATA Statement

The **DATA** statement assigns initial values to variables before program execution. For more information, see DATA in the *A to Z Reference.*

DIMENSION Attribute and Statement

The DIMENSION attribute specifies that an object is an array, and defines the shape of the array. For more information, see DIMENSION in the *A to Z Reference*.

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. For more information, see EQUIVALENCE in the *A to Z Reference*.

This section also discusses the following:

- Making Arrays Equivalent
- Making Substrings Equivalent
- EQUIVALENCE and COMMON Interaction

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 **EQUIVALENC**E 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 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. The following table shows how these statements align the arrays:

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 the above table:

```
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. The following table 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))
```

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 the above table:

```
DIMENSION B(2:4,1:4), A(2:3,1:4)
EQUIVALENCE(B(6), A(4))
```

Making Substrings Equivalent

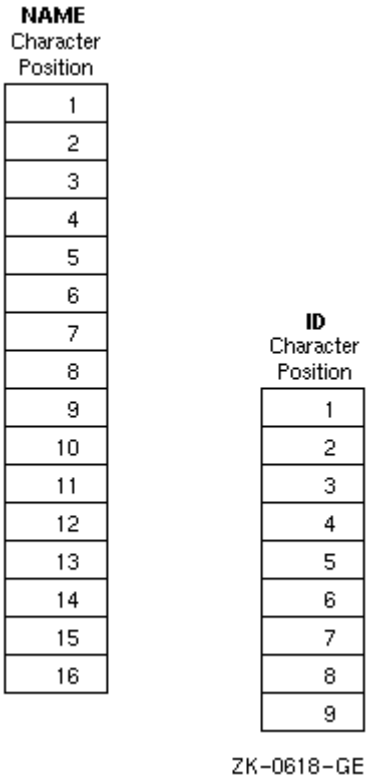
When you make one character substring equivalent to another character substring, the **EQUIVALENCE** statement also sets associations between the other corresponding characters in the character entities; for example:

```
CHARACTER NAME*16, ID*9
EQUIVALENCE(NAME(10:13), ID(2:5))
```

These statements cause character variables NAME and ID to share space (see the following figure). The arrays can also be aligned by the following statement:

```
EQUIVALENCE(NAME(9:9), ID(1:1))
```

Equivalence of Substrings

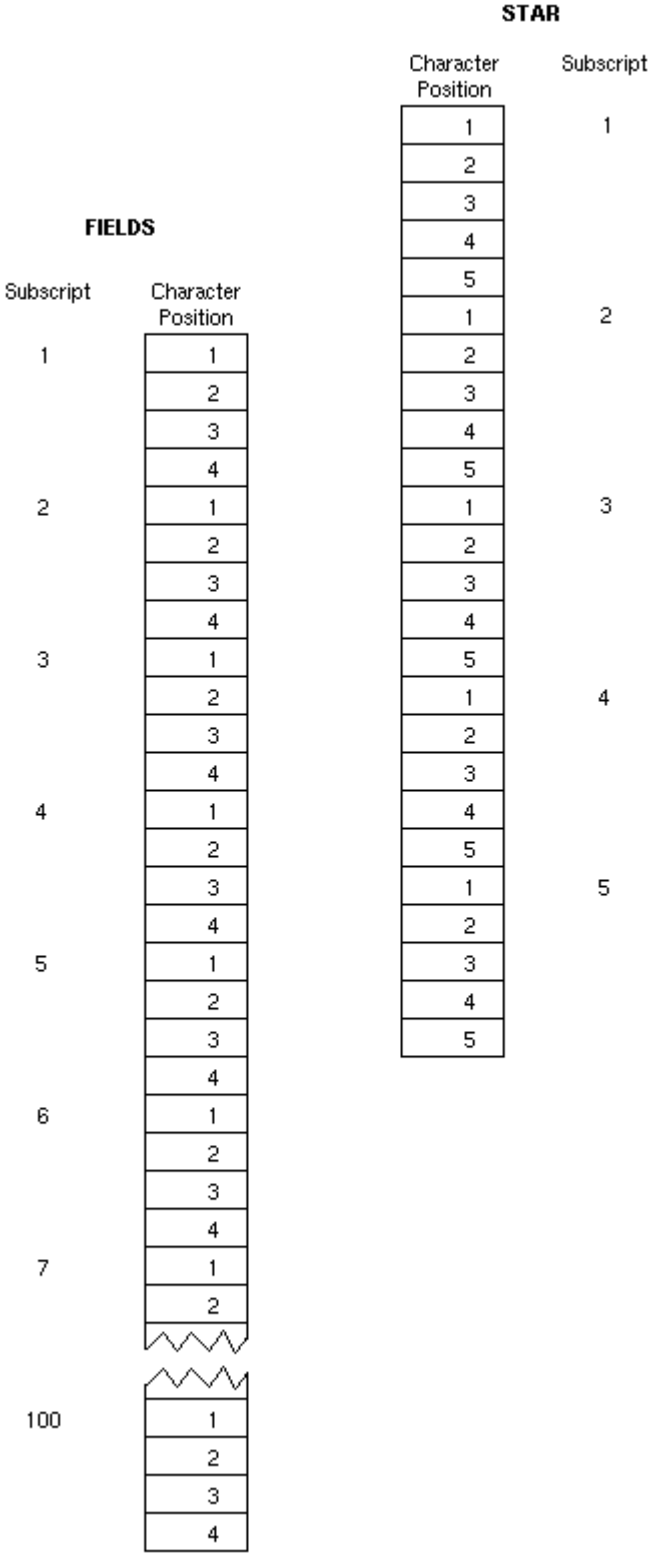


If the character substring references are array elements, the **EQUIVALENCE** statement sets associations between the other corresponding characters in the complete arrays.

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 the following figure).

```
CHARACTER FIELDS(100)*4, STAR(5)*5
EQUIVALENCE(FIELDS(1)(2:4), STAR(2)(3:5))
```

Equivalence of Character Arrays


STAR
Character Position
Subscript
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
FIELDS
Subscript
Character Position
1
2
3
4
1
2
3
4
1
2
3
4
1
2
3
4
1
2
3
4
1
2
3
4
1
2
3
4
1
2
3
4
100
1
2
3
4

ZK-0619-GE

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.

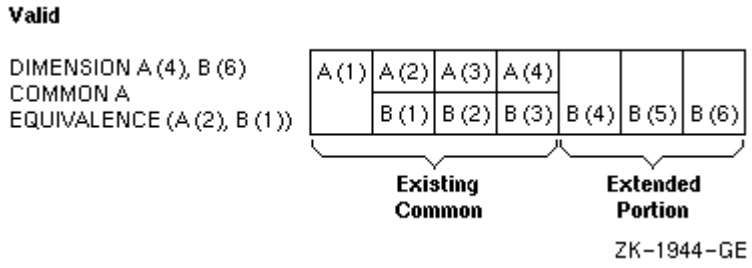
EQUIVALENCE and COMMON Interaction

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.

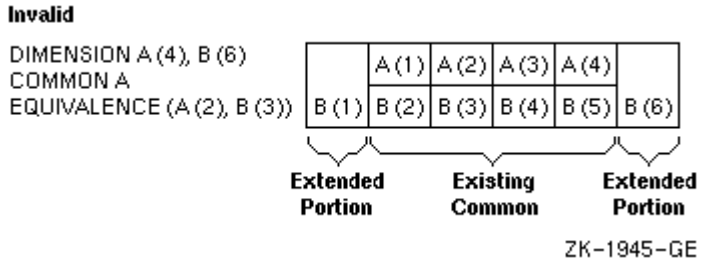
Examples

The following two figures demonstrate valid and invalid extensions of the common block, respectively.

A Valid Extension of a Common Block



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.
```

EXTERNAL Attribute and Statement

The EXTERNAL attribute allows an external or dummy procedure to be used as an actual argument. For more information, see EXTERNAL in the *A to Z Reference*.

IMPLICIT Statement

The **IMPLICIT** statement overrides the default implicit typing rules for names. For more information, see IMPLICIT in the *A to Z Reference*.

INTENT Attribute and Statement

The INTENT attribute specifies the intended use of one or more dummy arguments. For more information, see INTENT in the *A to Z Reference*.

INTRINSIC Attribute and Statement

The INTRINSIC attribute allows the specific name of an intrinsic procedure to be used as an actual argument. Certain specific function names cannot be used; these are indicated in Functions Not Allowed as Actual Arguments in the *A to Z Reference*.

For more information, see INTRINSIC in the *A to Z Reference*.

NAMELIST Statement

The **NAMELIST** statement associates a name with a list of variables. This group name can be referenced in some input/output operations. For more information, see NAMELIST in the *A to Z Reference*.

OPTIONAL Attribute and Statement

The OPTIONAL attribute permits dummy arguments to be omitted in a procedure reference. For more information, see OPTIONAL in the *A to Z Reference*.

PARAMETER Attribute and Statement

The PARAMETER attribute defines a named constant. For more information, see PARAMETER in the *A to Z Reference*.

POINTER Attribute and Statement

The POINTER attribute specifies that an object is a pointer (a dynamic variable). For more information, see POINTER in the *A to Z Reference*.

PUBLIC and PRIVATE Attributes and Statements

The PRIVATE and PUBLIC attributes specify the accessibility of entities in a module. (These attributes are also called accessibility attributes.) For more information, see PUBLIC and PRIVATE in the *A to Z Reference*.

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. For more information, see SAVE in the *A to Z Reference*.

TARGET Attribute and Statement

The TARGET attribute specifies that an object can become the target of a pointer. For more information, see TARGET in the *A to Z Reference*.

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. For more information, see VOLATILE in the *A to Z Reference*.

Dynamic Allocation

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 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.

Note: Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. Dynamic allocations that are too large or otherwise attempt to use the protected memory of other applications result in General Protection Fault errors. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Control Panel or redefine the swap file size.

Some programming techniques can help minimize memory requirements, such as using one large array instead of two or more individual arrays. Allocated arrays that are no longer needed should be deallocated.

This chapter contains information on the following topics:

For More Information:

- See Pointer Assignments.
- See Automatic arrays.
- See the NULL intrinsic function, which can also be used to disassociate a pointer.

ALLOCATE Statement

The **ALLOCATE** statement dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized. For more information, see ALLOCATE in the *A to Z Reference*.

This section also discusses the following:

- Allocation of Allocatable Arrays
- Allocation of Pointer Targets

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

The following example shows a program that performs virtual memory allocation. This program uses Fortran 90 standard-conforming statements instead of calling an operating system memory allocation routine.

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:

- See ALLOCATED.
- See ALLOCATE.

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:

- See POINTER.
- See Pointer assignments.
- See ASSOCIATED.

DEALLOCATE Statement

The **DEALLOCATE** statement frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated). For more information, see DEALLOCATE in the *A to Z Reference*.

This section also discusses the following:

- Deallocation of Allocatable Arrays
- Deallocation of Pointer Targets

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:

- See Host association.
- See TARGET.
- See RETURN.
- See END.
- See SAVE.

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:

- See POINTER.
- See COMMON.
- See NULL.
- See Host association.
- See TARGET.
- See RETURN.
- See END.
- See SAVE.

NULLIFY Statement

The **NULLIFY** statement disassociates a pointer from its target. For more information, see NULLIFY in the *A to Z Reference*.

Execution Control

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 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.

This chapter contains information on the following topics:

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, but this is an obsolescent feature in Fortran 90 (see Obsolescent Language Features in Fortran 90).

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:

- See IF constructs.
- See CASE constructs.
- See DO constructs.

Unconditional GO TO Statement

The unconditional **GO TO** statement transfers control to the same branch target statement every time it executes. For more information, see GOTO -- Unconditional in the *A to Z Reference*.

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. For more information, see GOTO -- Computed in the *A to Z Reference*.

The 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. For more information, see ASSIGN and GOTO -- Assigned in the *A to Z Reference*.

Arithmetic IF Statement

The arithmetic **IF** statement conditionally transfers control to one of three statements, based on the value of an arithmetic expression. For more information, see IF -- Arithmetic in the *A to Z Reference*.

CALL Statement

The **CALL** statement transfers control to a subroutine subprogram. For more information, see CALL in the *A to Z Reference*.

CASE Constructs

The **CASE** construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a **SELECT CASE** statement. For more information, see CASE in the *A to Z Reference*.

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. For more information, see CONTINUE in the *A to Z Reference*.

DO Constructs

The **DO** construct controls the repeated execution of a block of statements or constructs. For more information, see DO in the *A to Z Reference*.

This section also discusses the following topics:

- Forms for DO Constructs
- Execution of DO Constructs
- DO WHILE Statement
- CYCLE Statement
- EXIT Statement

Forms for DO Constructs

A **DO** construct can be in block or nonblock form. For more information, see DO in the *A to Z Reference*.

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:

- If there is no loop control (a simple **DO** construct), statements in the **DO** range are repeated until the **DO** statement is terminated explicitly by a statement within the range.

- If loop control is a **DO WHILE** statement, 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, see the DO WHILE statement.)

- If 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, see Iteration Loop Control.)

This section also discusses the following topics:

- Nested DO Constructs
- Extended Range

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 an obsolescent feature in Fortran 90, which has been deleted in Fortran 95. DIGITAL 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. (The compiler option /f66 can affect this.) 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:

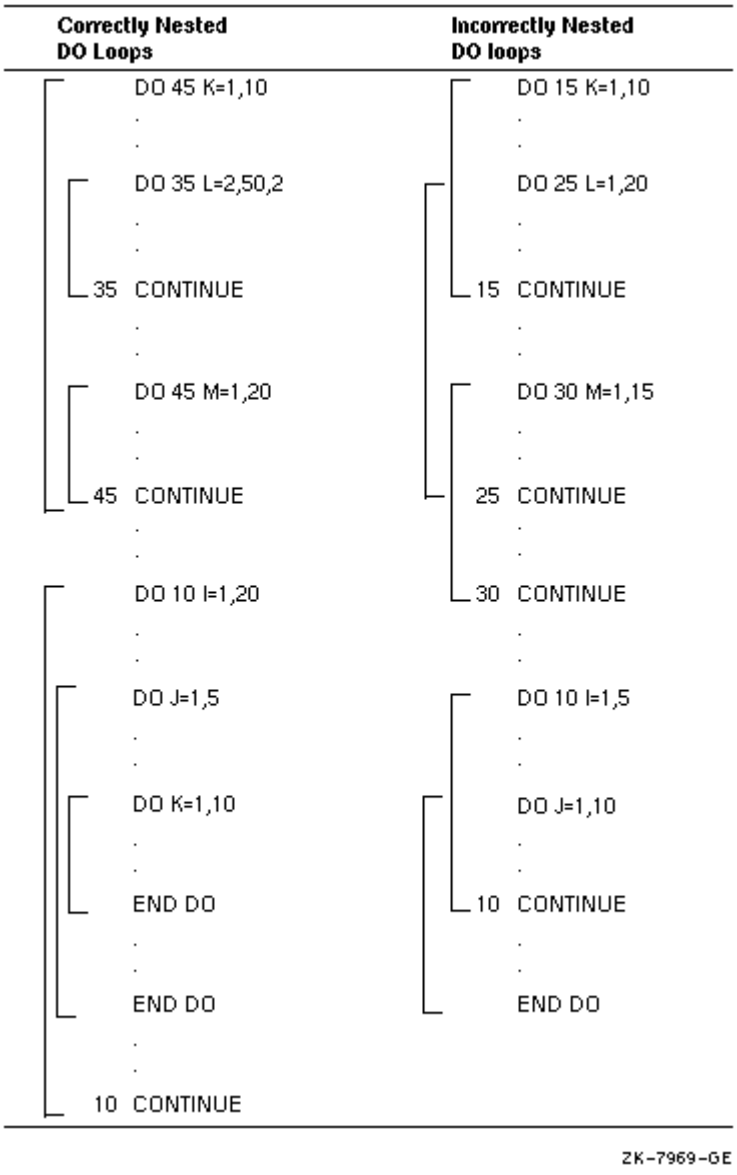
For details on obsolescent features in Fortran 90 and Fortran 95, as well as features deleted in Fortran 95, see Obsolescent and Deleted Language Features.

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.

The following figure shows correctly and incorrectly nested **DO** constructs:

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.

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.

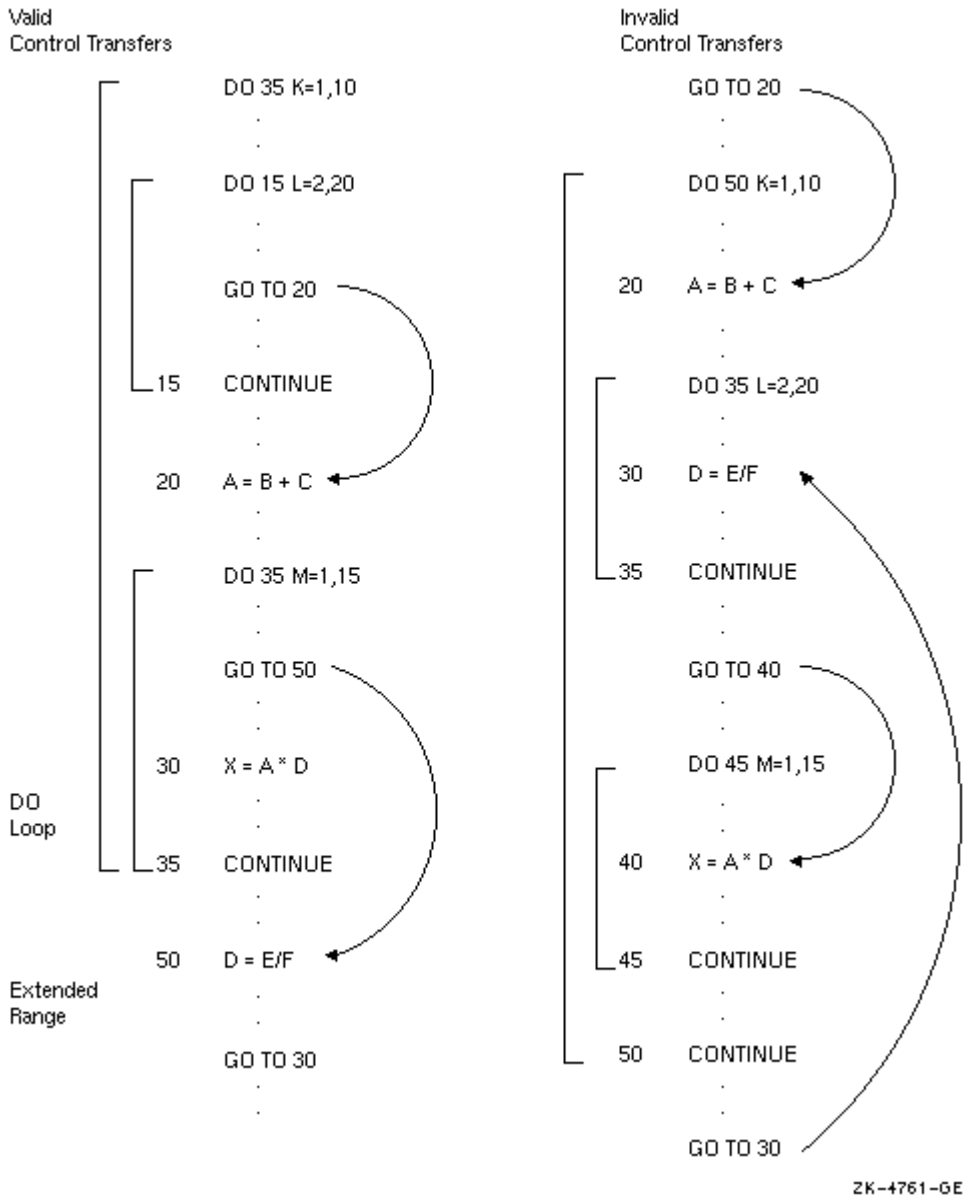
The range of the construct is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the construct.

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.

The following figure shows valid and invalid extended range control transfers:

Control Transfers and Extended Range



DO WHILE Statement

The **DO WHILE** statement executes the range of a **DO** construct while a specified condition remains true. For more information, see DO WHILE in the *A to Z Reference*.

CYCLE Statement

The **CYCLE** statement interrupts the current execution cycle of the innermost (or named) **DO** construct. For more information, see CYCLE in the *A to Z Reference*.

EXIT Statement

The **EXIT** statement terminates execution of a **DO** construct. For more information, see EXIT in the

A to Z Reference.

END Statement

The **END** statement marks the end of a program unit. For more information, see END in the *A to Z Reference*.

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:

See Arithmetic IF Statement.

IF Construct

The **IF** construct conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. For more information, see IF construct in the *A to Z Reference*.

IF Statement

The **IF** statement conditionally executes one statement based on the value of a logical expression. For more information, see IF -- Logical in the *A to Z Reference*.

PAUSE Statement

The **PAUSE** statement temporarily suspends program execution until the user or system resumes execution. For more information, see PAUSE in the *A to Z Reference*.

For alternate methods of pausing while reading from and writing to a device, see READ and WRITE in the *A to Z Reference*.

RETURN Statement

The **RETURN** statement transfers control from a subprogram to the calling program unit. For more information, see RETURN in the *A to Z Reference*.

STOP Statement

The **STOP** statement terminates program execution before the end of the program unit. For more information, see STOP in the *A to Z Reference*.

Program Units and Procedures

A Fortran 90 program consists of one or more program units. The principal kinds of program units are: the main program, external subprograms (user-written functions and subroutines), modules, and block data program units.

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	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

A *function* is invoked in an expression and 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, and does not return a particular value.

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.

This chapter contains information on the following topics:

- Argument association
- Procedure interfaces
- The CONTAINS statement
- The ENTRY statement

For More Information:

- See Program structure.
- See Intrinsic procedures.
- See Scope.
- See RECURSIVE.

Main Program

A main program is a program unit whose first statement is not a **SUBROUTINE**, **FUNCTION**, **MODULE**, or **BLOCK DATA** statement. Program execution always begins with the first executable statement in the main program, so there must be exactly one main program unit in every executable program. For more information, see PROGRAM in the *A to Z Reference*.

Modules and Module Procedures

A module program unit contains specifications and definitions that can be made accessible to other program units. For the module to be accessible, the other program units must reference the module in a **USE** statement, and the module entities must be public. For more information, see MODULE in the *A to Z Reference*.

A module procedure is a procedure declared and defined in a module, between its **CONTAINS** and **END** statements. For more information, see MODULE PROCEDURE in the *A to Z Reference*.

This section also discusses:

- Module References
- USE Statement

Module References

A module is referenced in a **USE** statement. If a module reference appears in a program unit, the program unit can 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:

- See the USE statement.
- See the PRIVATE and PUBLIC attributes.
- See Use association.

USE Statement

The **USE** statement gives a program unit accessibility to public entities in a module. For more information, see USE in the *A to Z Reference*.

Examples

Entities in modules can be accessed either through their given name, or through aliases declared in the **USE** statement of the main program unit. For example:

```
USE MODULE_LIB, XTABS => CROSSTABS
```

This statement accesses the routine called `CROSSTABS` in `MODULE_LIB` by the name `XTABS`. This way, if two modules have routines called `CROSSTABS`, one program can use them both simultaneously by assigning a local name in its **USE** statement.

When a program or subprogram renames a module entity, the local name (`XTABS`, in the preceding example) is accessible throughout the scope of the program unit that names it.

The ONLY option also allows public variables to be renamed. Consider the following:

```
USE MODULE_A, ONLY: VARIABLE_A => VAR_A
```

In this case, the host program accesses only `VAR_A` from module `A`, and refers to it by the name `VARIABLE_A`.

Consider the following example:

```
MODULE FOO
   integer foos_integer
 PRIVATE
   integer foos_secret_integer
END MODULE FOO
```

PRIVATE, in this case, makes the PRIVATE attribute the default for the entire module `FOO`. To make `foos_integer` accessible to other program units, add the line:

```
PUBLIC :: foos_integer
```

Alternatively, to make only `foos_secret_integer` inaccessible outside the module, rewrite the module as follows:

```
MODULE FOO
  integer foos_integer
  integer, private::foos_secret_integer
END MODULE FOO
```

Block Data Program Units

A block data program unit provides initial values for nonpointer variables in named common blocks. For more information, see BLOCK DATA in the *A to Z Reference*.

Examples

An example of a block data program unit follows:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10,10)
   DATA A /1.0/, B /2.0/, C /100*0.0/
END BLOCK DATA WORK
```

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 Defining Generic Operators).
[2] A subroutine can also be invoked by a defined assignment (see Defining Generic Assignment).

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 which is used to evaluate the expression.

The following topics are discussed in this section:

- General rules for function and subroutine subprograms
- Functions
- Subroutines
- Statement functions

For More Information:

- See the ENTRY statement.
- See the CALL statement.

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.

For More Information:

- See RECURSIVE.
- See PURE procedures.
- See user-defined ELEMENTAL procedures.
- See Module procedures.
- See Internal procedures.
- See External procedures.
- See Optional arguments.
- See INTENT.

Recursion

A recursive procedure is a function or subroutine that references itself, either directly or indirectly. For more information, see RECURSIVE in the *A to Z Reference*.

Pure Procedures

A pure procedure is a user-defined procedure that has no side effects. Pure procedures are a feature of Fortran 95.

For more information, see PURE in the *A to Z Reference*.

Elemental Procedures

An elemental procedure is a user-defined procedure that is a restricted form of pure procedure. For more information, see PURE and ELEMENTAL in the *A to Z Reference*.

Functions

A *function* subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression. For more information, see FUNCTION in the *A to Z Reference*.

This section also discusses the following:

- The RESULT Keyword
- Function References

RESULT Keyword

If you use the **RESULT** keyword in a **FUNCTION** statement, you can specify a local variable name for the function result. For more information, see RESULT in the *A to Z Reference*.

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 Functions Not Allowed as Actual Arguments).

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:

- See the INTENT attribute.
- See Defining Generic Operators.
- See Dummy Procedure Arguments.
- See Intrinsic Procedures.
- See Optional arguments.
- See the RESULT keyword.
- See the FUNCTION statement.
- On procedure arguments, see Argument Association.

Subroutines

A *subroutine* subprogram is invoked in a **CALL** statement or by a defined assignment statement, and does not return a particular value. For more information, see SUBROUTINE in the *A to Z Reference*.

Statement Functions

A statement function is a procedure defined by a single statement in the same program unit in which the procedure is referenced. For more information, see Statement Function in the *A to Z Reference*.

External Procedures

External procedures (functions or subroutines) are defined by external subprograms that are not part of any other program unit. External procedures can also be defined by means other than Fortran 90.

External procedures can be invoked by the main program or any procedure of an executable program, and they can be optionally preceded by an **OPTIONS** statement.

In Fortran 90, external procedures can include internal subprograms (defining internal procedures). An internal subprogram begins with a **CONTAINS** statement.

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:

- See Functions, Subroutines, and Statement Functions.
- See Procedure Interfaces.
- On passing arguments, see your programmer's guide.

Internal Procedures

An internal procedure is defined by an internal subprogram. Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram. The program unit in which the internal procedure appears is called its host.

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:

- An internal procedure is local to its host.
- An internal procedure has access to host entities by host association.
- An internal procedure must not be argument-associated with a dummy 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.

Examples

The following example shows an internal procedure:

```
PROGRAM COLOR_GUIDE
...
CONTAINS
  FUNCTION HUE(BLUE)   ! An internal procedure
  ...
  END FUNCTION HUE
END PROGRAM
```

The following example program contains an internal subroutine `find`, which performs calculations that the main program then prints. The variables `a`, `b`, and `c` declared in the host program are also known to the internal subroutine.

```
 program INTERNAL
 ! shows use of internal subroutine and CONTAINS statement
    real a,b,c
    call find
    print *, c
 contains
    subroutine find
     read *, a,b
     c = sqrt(a**2 + b**2)
    end subroutine find
 end
```

For More Information:

- See Functions, Subroutines, and Statement Functions.
- See Host association.
- See Procedure Interfaces.
- See CONTAINS.

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
- The different kinds of arguments:
  - Array arguments
  - Pointer arguments
  - Assumed-length character arguments
  - Character constant and Hollerith arguments
  - Alternate return arguments
  - Dummy procedure arguments
- References to generic procedures
- References to non-Fortran procedures (**%DESCR %REF**, **%VAL**, and **%LOC**)

For More Information:

- On argument keywords in subroutine references, see CALL.
- On argument keywords in function references, see Function References.

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.

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.

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.

Optional arguments must have explicit procedure interfaces so that appropriate argument associations can be made.

The **PRESENT** intrinsic function can be used to determine if an actual argument is associated with an optional dummy argument in a particular reference.

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.

The following shows another example:

```
! Arguments can be passed out of order, but must be
!  associated with the correct dummy argument.
CALL EXT1 (Z=C, X=A, Y=B)
. . .
END

SUBROUTINE EXT1(X,Y,Z)
   REAL X, Y
   REAL, OPTIONAL :: Z
   . . .
END SUBROUTINE
```

In this case, argument A is associated with dummy argument X by explicit assignment. Once EXT1 executes and returns, A is no longer associated with X, B is no longer associated with Y, and C is no longer associated with Z.

For More Information:

- See the OPTIONAL attribute.

- See the PRESENT intrinsic function.
- On general rules for procedure argument association, see Argument association.
- On argument keywords in subroutine references, see CALL.
- On argument keywords in function references, see Function References.

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 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:

- See Arrays.
- See Array association.
- On general rules for procedure argument association, see Argument association.
- On array element order, see Array Elements.
- On explicit-shape arrays, see Explicit-Shape Specifications.
- On assumed-shape arrays, see Assumed-Shape Specifications.
- On assumed-size arrays, see Assumed-Size Specifications.

Pointer Arguments

An argument is a pointer if it is declared with the POINTER attribute.

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.

If both the dummy and actual arguments are pointers, an explicit interface is required.

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.

The pointer association status of the dummy argument can change during the execution of the procedure, and any such changes are reflected in the actual argument.

If a pointer actual argument is an array, the corresponding pointer dummy argument must be a deferred-shape array.

A pointer actual argument can correspond to a nonpointer dummy argument if the actual argument is associated.

If the actual argument has the TARGET attribute, any pointers associated with it do not become associated with the corresponding dummy argument when the procedure is invoked, but remain associated with the actual argument.

If the dummy argument has the TARGET attribute, any pointer associated with it becomes undefined when execution of the procedure completes.

For More Information:

- See POINTER.
- See Pointer assignments.
- See the TARGET attribute.
- On general rules for procedure argument association, see Argument association.

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:

- See the LEN intrinsic function.
- On general rules for procedure argument association, see Argument association.

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:

For details on general rules for procedure argument association, see Argument association.

Alternate Return Arguments

Alternate return dummy arguments can appear in a subroutine argument list to transfer control to

other statements. The alternate return is indicated by an asterisk (*). (An alternate return is an obsolescent feature in Fortran 90 and Fortran 95.)

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 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 Argument association.
- See the SUBROUTINE statement.
- See the CALL statement.
- See the RETURN statement.
- On obsolescent features in Fortran 90 and Fortran 95, see Obsolescent and Deleted Language Features.

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 Functions Not Allowed as Actual Arguments.)

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:

For details on general rules for procedure argument association, see Argument association.

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 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:

- See Unambiguous Generic Procedure References.
- See Intrinsic procedures.
- On the rules for resolving ambiguous procedure references, see Resolving Procedure References.
- On user-defined generic procedures, see Defining Generic Names for Procedures.

References to Generic Intrinsic Functions

The generic intrinsic function name **COS** lists four specific intrinsic functions that calculate cosines:

COS, **DCOS**, **QCOS**, **CCOS**, and **CDCOS**. These functions return different values: REAL(4), REAL(8), REAL(16) (VMS, U*X), COMPLEX(4), and COMPLEX(8), 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 Functions Not Allowed as Actual Arguments.)

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 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

The following example shows the local and global properties of an intrinsic function name. It uses intrinsic function **SIN** as follows:

- The name of a statement function
- The generic name of an intrinsic function
- The specific name of an intrinsic function
- The name of a user-defined function

Using and Redefining an Intrinsic Function Name

```
!     Compare ways of computing sine

      PROGRAM SINES
        DOUBLE PRECISION X, PI
        PARAMETER (PI=3.14159265358979238D0)
        COMMON V(3)

!     Define SIN as a statement function 1

        SIN(X) = COS(PI/2-X)
        DO X = -PI, PI, 2*PI/100

!     Reference the statement function SIN 2

          WRITE (6,100) X, V, SIN(X)
        END DO
        CALL COMPUT(X)
100     FORMAT (5F10.7)
      END

      SUBROUTINE COMPUT(Y)
        DOUBLE PRECISION Y

!     Use intrinsic function SIN as an actual argument 3

        INTRINSIC SIN
        COMMON V(3)

!     Define generic reference to double-precision sine 4

        V(1) = SIN(Y)

!     Use intrinsic function SIN as an actual argument 5

        CALL SUB(REAL(Y),SIN)
      END

      SUBROUTINE SUB(A,S)

!     Declare SIN as name of a user function 6

        EXTERNAL SIN


!     Declare SIN as type DOUBLE PRECISION 7

        DOUBLE PRECISION SIN
        COMMON V(3)
```

```
!     Evaluate intrinsic function SIN 8

        V(2) = S(A)

!     Evaluate user-defined SIN function 9

        V(3) = SIN(A)
      END

!     Define the user SIN function 10

      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
```

1 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.

2 The statement function SIN is called.

3 The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at 5.

4 The generic function name SIN is used to refer to the double-precision sine function.

5 The single-precision intrinsic sine function is used as an actual argument.

6 The name SIN is declared a user-defined function name.

7 The type of SIN is declared double precision.

8 The single-precision sine function passed at 5 is evaluated.

9 The user-defined SIN function is evaluated.

10 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:

- See the EXTERNAL attribute.
- See the INTRINSIC attribute.
- See Intrinsic procedures.

- On the scope of names, see Names.

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.

Examples

Consider the following:

```
REAL, DIMENSION (2) :: a, b
a(1) = 4; a(2) = 9
b = SQRT(a)                ! sets b(1) = SQRT(a(1)), and b(2) = SQRT(a(2))
```

For More Information:

- See Arrays.
- On elemental procedures, see Intrinsic Procedures.

References to Non-Fortran Procedures

To facilitate references to non-Fortran procedures, DIGITAL Fortran provides the following built-in functions:

- To pass actual arguments:
  - %REF
  - %VAL
  - %DESCR (VMS only)

- To compute the internal address of a storage item
  - %LOC

Procedure Interfaces

A procedure interface specifies the name and characteristics of a procedure, the name and attributes 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. 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.

Examples

An example of an interface block follows:

```
INTERFACE
  SUBROUTINE Ext1 (x, y, z)
  REAL, DIMENSION (100,100) :: x, y, z
  END SUBROUTINE Ext1
```

```
  SUBROUTINE Ext2 (x, z)
  REAL x
  COMPLEX (KIND = 2) z (2000)
  END SUBROUTINE Ext2

  FUNCTION Ext3 (p, q)
  LOGICAL Ext3
  INTEGER p (1000)
  LOGICAL q (1000)
  END FUNCTION Ext3
END INTERFACE
```

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:

- See Optional arguments.
- See Array arguments.
- See Pointer arguments.
- On argument keywords in subroutine references, see CALL.
- On argument keywords in function references, see Function references.
- On user-defined generic procedures, see Defining Generic Names for Procedures.
- On defined operators, see Defining Generic Operators.
- On defined assignment, see Defining Generic Assignment.

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.

For more information on interface blocks, see INTERFACE in the *A to Z Reference*.

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 Unambiguous Generic Procedure References.

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)
```

Consider the following:

```
 INTERFACE LINE_EQUATION

   SUBROUTINE REAL_LINE_EQ(X1,Y1,X2,Y2,M,B)
     REAL,INTENT(IN) :: X1,Y1,X2,Y2
     REAL,INTENT(OUT) :: M,B
   END SUBROUTINE REAL_LINE_EQ

   SUBROUTINE INT_LINE_EQ(X1,Y1,X2,Y2,M,B)
     INTEGER,INTENT(IN) :: X1,Y1,X2,Y2
     INTEGER,INTENT(OUT) :: M,B
   END SUBROUTINE INT_LINE_EQ

 END INTERFACE
```

In this example, LINE_EQUATION is the generic name which can be used for either REAL_LINE_EQ or INT_LINE_EQ. Fortran selects the appropriate subroutine according to the nature of the arguments passed to LINE_EQUATION. Even when a generic name exists, you can always invoke a procedure by its specific name. In the previous example, you can call REAL_LINE_EQ by its specific name (REAL_LINE_EQ), or its generic name LINE_EQUATION.

For More Information:

For details on interface blocks, see INTERFACE.

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:

- See INTENT.
- On interface blocks, see INTERFACE.
- On intrinsic operators, see Expressions.
- On defined operators and operations, see Defined Operations.

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 Unambiguous Generic Procedure References.

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:

- See Defined Assignments.
- See INTENT.
- On interface blocks, see INTERFACE.

CONTAINS Statement

For information on the **CONTAINS** statement, see CONTAINS in the *A to Z Reference.*

ENTRY Statement

The **ENTRY** statement provides multiple entry points within a subprogram. It is not executable and must precede any **CONTAINS** statement (if any) within the subprogram. For more information, see ENTRY in the *A to Z Reference*.

This section also discusses:

- ENTRY Statements in Function Subprograms
- ENTRY Statements in Subroutine Subprograms

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) (VMS, U*X)
Group	Type default character (with identical lengths)

3	

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:

See the RESULT keyword.

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.

Intrinsic Procedures

Intrinsic procedures are functions and subroutines that are included in the Fortran 90 library. There are four classes of these intrinsic procedures, as follows:

- 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.

  Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays. For example, consider the following:

```
a = b + c
...                ! a, b, c, and s are all arrays of similar shape
s = sum(a)
```

  The above statements can replace entire **DO** loops.

  Consider the following:

```
real, dimension (5,5) x,y
. . .      !Assign values to x.
y = sin(x) !Pass the entire array as an argument.
```

  In this example, since the **SIN**(X) function is an elemental procedure, it operates element-by-element on the array x when you pass it the name of the whole array.

- 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. Not all specific intrinsic functions are allowed as actual arguments in all circumstances. Functions Not Allowed as Actual Arguments lists specific functions that cannot be passed as actual arguments.

This chapter also contains information on the following topics:

- Argument keywords in intrinsic procedures
- Overview of intrinsic procedures

The A to Z Reference contains the descriptions of all intrinsics listed in alphabetical order. Each reference entry indicates whether the procedure is inquiry, elemental, transformational, or nonelemental, and whether it is a function or a subroutine.

For More Information:

- See Argument association.
- See the MERGE intrinsic function.
- See Optional arguments.
- See Language Summary Tables.
- See Data representation models.
- On generic intrinsic procedures, see References to Generic Intrinsic Functions.
- On elemental references to intrinsic procedures, see References to Elemental Intrinsic Procedures.

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)
	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.

For More Information:

- On argument keywords in subroutine references, see CALL.
- On argument keywords in function references, see Function references.

Overview 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 the A to Z Reference.

Categories of Intrinsic Functions

Generic intrinsic functions can be divided into categories, as shown in the following table:

Categories of Intrinsic Functions

Category	Subcategory	Description
Numeric	Computation	Elemental functions that perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DPROD, FLOAT, FLOOR, IFIX, IMAG, INT, MAX, MAX1, MIN, MIN1, MOD, MODULO, NINT, QEXT, QFLOAT, RAN, REAL, SIGN, SNGL, ZEXT

	Manipulation[1]	Elemental functions that 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]	Functions that 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	Functions that perform vector and matrix multiplication: DOT_PRODUCT, MATMUL
	System	Functions that return information about a process or processor: PROCESSORS_SHAPE, NWORKERS, NUMBER_OF_PROCESSORS, SECNDS
Kind type		Functions that return kind type parameters: KIND, SELECTED_INT_KIND, SELECTED_ REAL_KIND
Mathematical		Functions that 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	Elemental functions that 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	Function that lets you determine parameter *s* (the bit size) in the bit model[4]: BIT_SIZE
	Representation	Functions that return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ
Character	Comparison	Elemental functions that make a lexical comparison of the character-string arguments and return a default logical result: LGE, LGT, LLE, LLT
	Conversion	Elemental functions that take character arguments and return integer, ASCII, or character values[3]: ACHAR, CHAR, IACHAR, ICHAR
	String handling	Functions that perform operations on character strings, return

		lengths of arguments, and search for certain arguments: ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, REPEAT, SCAN, TRIM, VERIFY
	Inquiry	Function that returns length of argument: LEN
Array	Construction	Functions that construct new arrays from the elements of existing array: MERGE, PACK, SPREAD, UNPACK
	Inquiry	Functions that 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	Transformational functions that find the geometric locations of the maximum and minimum values of an array: MAXLOC, MINLOC
	Manipulation	Transformational functions that shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE
	Reduction	Transformational functions that 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		Functions that do the following: • Check for argument presence (PRESENT) • Check for pointer association (ASSOCIATED) • Return a logical value of an argument (LOGICAL) • Convert a bit pattern (TRANSFER) • Return the class of a floating-point argument (FP_CLASS) • Test for Not-a-Number values (ISNAN) • Count actual arguments passed to a routine (IARGCOUNT) • Return a pointer to an actual argument list for a routine (IARGPTR) • Return the internal address of a storage item (LOC) • Check for end-of-file (EOF) • Allocate memory (MALLOC) • Return a disassociated pointer (NULL) • Let you use assembler instructions in an executable program (ASM) • Return the upper 64 bits of a 128-bit unsigned result (MULT_HIGH)

[1] All of the numeric manipulation, and many of the numeric inquiry functions are defined by the model sets for integers and reals.
[2] The value of the argument does not have to be defined.
[3] The DIGITAL Fortran processor character set is ASCII, so ACHAR = CHAR and IACHAR = ICHAR.
[4] For more information on bit functions, see Bit functions.

The following table summarizes the generic intrinsic functions and indicates whether they are elemental, inquiry, or transformational functions, if applicable. Optional arguments are shown within square brackets.

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
ASIND (X)	E	The arc sine (in degrees) of the argument
ASM (STRING [,A,...) [1]	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 (in radians) of the argument
COSD (X)	E	The cosine (in degrees) of the argument
COSH (X)	E	The hyperbolic cosine of the argument
COTAN (X)	E	The cotangent (in radians) of the argument
COTAND (X)	E	The cotangent (in degrees) of the argument
COUNT (MASK [,DIM])	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 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)
DPROD (X, Y)	E	The double precision product of two real arguments
EOF (A)	I	.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record
EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])	T	An array that has the elements of the argument array end-off shifted
EPSILON (X)	I	The number that is almost negligible when compared to one
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
IARGCOUNT ( )[2]	I	The count of actual arguments passed to the current routine.
IARGPTR ( )	I	A pointer to the actual argument list for the current routine.
IBCLR (I, POS)	E	The specified position of argument I cleared (set to zero)
IBCHNG (I, POS)	E	The reversed value of a specified bit
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])	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
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])	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)	I	The length (number of characters) of the argument character string
LEN_TRIM (STRING)	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
MALLOC (I)	E	The starting address for the block of memory allocated
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
MAXLOC (ARRAY [,DIM] [,MASK])	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])	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)
MULT_HIGH (I, J) [1]	E	The upper (leftmost) 64 bits of the 128-bit unsigned result
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
NUMBER_OF_PROCESSORS ( )	I	The total number of processors (peers) available to the program
NWORKERS ( ) [3]	I	The number of executing processes
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
PROCESSORS_SHAPE ( )	I	The shape of an implementation-dependent hardware processor array
PRODUCT (ARRAY [,DIM] [,MASK])	T	The product of the elements of the argument array
QEXT (A)[4]	E	The corresponding REAL(16) precision value of the argument.
QFLOAT (A)[4]	E	The corresponding REAL(16) precision value of the integer argument.
RADIX (X)	I	The base of the model for the argument
RAN (I)	N	The next number from a sequence of pseudorandom numbers (uniformly distributed in the range 0 to 1)
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])	E	The position of the specified character (or set of characters) within a string
SECNDS (X)	E	The system time of day (or elapsed time) as a

		floating-point value in seconds
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 of the exponent part (of the model for the argument) set to a specified value
SHAPE (SOURCE)	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 (in radians) of the argument
SIND (X)	E	The sine (in degrees) of the argument
SINH (X)	E	The hyperbolic sine of the argument
SIZE (ARRAY [,DIM])	I	The size (total number of elements) of the argument array (or one of its dimensions)
SIZEOF (X)	I	The bytes of storage used by the argument
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
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 (in radians) of the argument
TAND (X)	E	The tangent (in degrees) of the argument
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])	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])	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)	E	A zero-extended value of the argument

[1] Alpha only
[2] VMS only
[3] Included for compatibility with older versions of DIGITAL Fortran 77.
[4] VMS, U*X

Key to Classes

E-Elemental
I-Inquiry
T-Transformational
N-Nonelemental

Intrinsic Subroutines

The following table lists the intrinsic subroutines. All these subroutines are nonelemental except for **MVBITS**.

Intrinsic Subroutines

Subroutine	Value Returned
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])	Image exit status is optionally returned; the program is

	terminated, all files closed, and control is returned 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]	A sequence of bits (bit field) is copied 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])	The initialization or retrieval of the pseudorandom number generator seed value
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.

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) (Alpha only) and LOGICAL(8) (Alpha only)
- 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 on Alpha processors; 0 to 31 on Intel processors.

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, DIGITAL Fortran provides the following integer data types:

Data Type	Storage Required (in bytes)
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)[1]	8
[1] Alpha only	

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.

Data Transfer I/O Statements

Input/Output (I/O) statements can be used for data transfer, file connection, file inquiry, and file positioning.

This section discusses data transfer and contains information on the following topics:

- An overview of records and files
- Components of data transfer statements

- Data transfer input statements:
  - READ statements
  - ACCEPT statements

- Data transfer output statements:
  - WRITE statements
  - PRINT and TYPE statements
  - REWRITE statements

File connection, file inquiry, and file positioning I/O statements are discussed in File Operation I/O Statements (WNT and W95).

See also Improve Overall I/O Performance in the *Programmer's Guide*.

Records and Files

A record is a sequence of values or a sequence of characters. There are three kinds of Fortran records, as follows:

- 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.

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 (VMS only), 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:

For details on formatted and unformatted data transfers and external file access methods, see your programmer's guide.

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 (VMS only)
[FMT=]format	END	IOSTAT	REC
[NML=]group	EOR	KEY[con] (VMS only)	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 Branch Specifiers.)

Following sections describe the I/O control list and I/O lists.

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 (VMS only)
- 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, follow the rules specified for the *nonkeyword form*.

The following sections describe the control-list specifiers in detail:

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.
- 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 as follows:

- On OpenVMS systems: STATUS='NEW'

- On DIGITAL UNIX, Windows NT, and Windows 95 systems: STATUS='UNKNOWN'

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:

- See the OPEN statement.
- On implicit logical assignments, see your programmer's guide.
- On using internal files, see your programmer's guide.

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:

- See the FORMAT statement.
- See the interaction between FORMAT statements and I/O lists.
- On list-directed input, see Rules for List-Directed Sequential READ Statements; output, see Rules for List-Directed Sequential WRITE Statements.

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:

For details on namelist input, see Rules for Namelist Sequential READ Statements; output, see Rules for Namelist Sequential WRITE Statements.

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:

See Alternative Syntax for a Record Specifier.

Key-Field-Value Specifier (VMS only)

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:

	In Ascending-Key Files	**In Descending-Key Files**
Specifier:	**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. [1] 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.

[1] Other collating sequences are available. For more information, see the *Guide to OpenVMS File Applications*.

Key-of-Reference Specifier (VMS only)

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:

See the key-field-value specifier.

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.

For More Information:

For details on the error numbers returned by IOSTAT, see your programmer's guide.

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 (VMS only), 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 a *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 (VMS only) 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 a *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 a *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 the IOSTAT specifier, see I/O Status Specifier and Using the IOSTAT Specifier and Fortran Exit Codes.
- On branch target statements, see Branch Statements and Using the END, EOR, and ERR Branch Specifiers.
- On error processing, see your programmer's guide.

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:

For details on advancing and nonadvancing I/O, see your programmer's guide.

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 default 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.

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.

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
```

The following shows more examples:

```
!  A variable and array element in iolist:
      REAL b(99)
      READ (*, 300) n, b(n) ! n and b(n) are the iolist
300   FORMAT (I2, F10.5) ! FORMAT statement telling what form the input data has

!  A derived type and type element in iolist:
      TYPE YOUR_DATA
        REAL a
        CHARACTER(30) info
        COMPLEX cx
      END TYPE YOUR_DATA
      TYPE (YOUR_DATA) yd1, yd2
      yd1.a = 2.3
      yd1.info = "This is a type demo."
      yd1.cx = (3.0, 4.0)
      yd2.cx = (4.5, 6.7)
!  The iolist follows the WRITE (*,500).
      WRITE (*, 500) yd1, yd2.cx
!  The format statement tells how the iolist will be output.
500   FORMAT (F5.3, A21, F5.2, ',', F5.2, ' yd2.cx = (', F5.2,
       ',',F5.2, ' )')
!  The output looks like:
!  2.300This is a type demo 3.00, 4.00 yd2.cx = ( 4.50, 6.70 )
```

The following example uses an array and an array section:

```
!  An array in the iolist:
      INTEGER handle(5)
      DATA handle / 5*0 /
      WRITE (*, 99) handle
99    FORMAT (5I5)
!  An array section in the iolist.
      WRITE (*, 100) handle(2:3)
100   FORMAT (2I5)
```

The following shows another example:

```
PRINT *,'(I5)', 2*3   ! The iolist is the expression 2*3.
```

The following example uses a namelist:

```
! Namelist I/O:
     INTEGER int1
     LOGICAL log1
     REAL r1
     CHARACTER (20) char20
     NAMELIST /mylist/ int1, log1, r1, char20
     int1 = 1
     log1 = .TRUE.
     r1 = 1.0
     char20 = 'NAMELIST demo'
     OPEN (UNIT = 4, FILE = 'MYFILE.DAT', DELIM = 'APOSTROPHE')
     WRITE (UNIT = 4, NML = mylist)
! Writes the following:
! &MYLIST
! INT1 = 1,
! LOG1 = T,
! R1 = 1.000000    ,
! CHAR20 = 'NAMELIST demo '
! /
     REWIND(4)
     READ (4, mylist)
```

For More Information:

For details on the general rules for I/O lists, see I/O Lists.

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* = *expr*1, *expr*2 [,*expr*3])

list
Is a list of variables, expressions, or constants (see Simple List Items in I/O Lists).

do-var
Is the name of a scalar integer or real variable. The variable must not be one of the input items or output 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.

Consider the following:

```
      INTEGER mydata(25)
      READ (10, 9000) (mydata(I), I=6,10,1)
9000  FORMAT (5I3)
```

In this example, the *iolist* specifies to put the input data into elements 6 through 10 of the array called mydata. The third value in the implied-**DO** loop, the increment, is optional. If you leave it out, the increment value defaults to 1.

For More Information:

- See DO constructs.
- On the general rules for I/O lists, see I/O Lists.

READ Statements

The **READ** statement is a data transfer input statement. Data can be input from external sequential, keyed-access (VMS only) or direct-access records, or from internal records. (For more information, see READ in the *A to Z Reference*.)

This section discusses the following topics:

- Forms for Sequential READ Statements
- Forms for Direct-Access READ Statements
- Forms and Rules for Internal READ Statements

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.

A sequential **READ** statement takes one of the following forms:

Formatted

READ (*eunit*, *format* [, *advance*] [, *size*] [, *iostat*] [, *err*] [, *end*] [, *eor*]) [*io-list*]

READ *form* [, *io-list*]

Formatted: List-Directed

READ (*eunit*, * [, *iostat*] [, *err*] [, *end*]) [*io-list*]
READ * [, *io-list*]

Formatted: Namelist

READ (*eunit*, *nml-group* [, *iostat*] [, *err*] [, *end*])
READ *nml*

Unformatted

READ (*eunit* [, *iostat*] [, *err*] [, *end*]) [*io-list*]

For more information, see READ in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Sequential READ Statements
- Rules for List-Directed Sequential READ Statements
- Rules for Namelist Sequential READ Statements
- Rules for Unformatted Sequential READ Statements

For More Information:

- See I/O control-list specifiers.
- See I/O lists.

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
```

For More Information:

- See READ.
- See Forms for Sequential READ Statements.

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.

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, 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 the table in Numeric Assignment Statements).

  - 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

The following example shows list-directed input and output:

```
REAL    a
INTEGER i
COMPLEX c
LOGICAL up, down
DATA a /2358.2E-8/, i /91585/, c /(705.60,819.60)/
DATA up /.TRUE./, down /.FALSE./
OPEN (UNIT = 9, FILE = 'listout', STATUS = 'NEW')
WRITE  (9, *) a, i
WRITE  (9, *) c, up, down
REWIND (9)
READ   (9, *) a, i
READ   (9, *) c, up, down
WRITE  (*, *) a, i
WRITE  (*, *) c, up, down
```

```
END
```

The preceding program produces the following output:

```
 2.3582001E-05     91585
(705.6000,819.6000) T F
```

For More Information:

- See READ.
- See Forms for Sequential READ Statements.
- On the literal constant forms of intrinsic data types, see Intrinsic Data Types.
- On list-directed output, see Rules for List-Directed Sequential WRITE Statements.

Rules for Namelist Sequential READ Statement

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 the table in Numeric Assignment Statements).

- 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 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.

The following shows another example:

```
      INTEGER a, b
      NAMELIST /mynml/ a, b
...
!  The following are all valid namelist variable assignments:
      &mynml a = 1 /
      $mynml a = 1 $
      $mynml a = 1 $end
      &mynml a = 1 &
      &mynml a = 1 $END
      &mynml
      a = 1
      b = 2
      /
```

For More Information:

- See the NAMELIST statement.
- See Rules for Formatted Sequential READ Statements.
- See an Alternative Form for Namelist External Records.
- On namelist output, see Rules for Namelist Sequential WRITE Statements.

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
```

For More Information:

- See READ.
- See Forms for Sequential READ Statements.

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:

Formatted

READ (*eunit*, *format*, *rec* [,*iostat*] [,*err*]) [*io-list*]

Unformatted

READ (*eunit*, *rec* [,*iostat*] [,*err*]) [*io-list*]

For more information, see READ in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Direct-Access READ Statements
- Rules for Unformatted Direct-Access READ Statements

For More Information:

- See I/O control-list specifiers.
- See I/O lists.
- On file sharing, see your programmer's guide.

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)
```

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, 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 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)
```

Forms for Indexed READ Statements (VMS only)

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:

Formatted

READ (*eunit, format, key* [,*keyid*] [,*iostat*] [,*err*]) [*io-list*]

Unformatted

READ (*eunit*, *key* [,*keyid*] [,*err*]) [*io-list*]

For more information, see READ in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Indexed READ Statements (VMS only)
- Rules for Unformatted Indexed READ Statements (VMS only)

For More Information:

- See I/O control-list specifiers.
- See I/O lists.

Rules for Formatted Indexed READ Statements (VMS only)

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.

Rules for Unformatted Indexed READ Statements (VMS only)

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.

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*]

For more information on syntax, see READ in the *A to Z Reference*.

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

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:

- See I/O control-list specifiers.
- See I/O lists.
- On list-directed input, see Rules for List-Directed Sequential READ Statements.
- On using internal files, see your programmer's guide.

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.

For more information, see ACCEPT in the *A to Z Reference*.

WRITE Statements

The **WRITE** statement is a data transfer output statement. Data can be output to external sequential, keyed-access (VMS only) or direct-access records, or to internal records. (For more information, see WRITE in the *A to Z Reference*.)

This section discusses the following topics:

- Forms for Sequential WRITE Statements
- Forms for Direct-Access WRITE Statements
- Forms and Rules for Internal WRITE Statements

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:

Formatted

WRITE (*eunit, format* [,*advance*] [,*iostat*] [,*err*]) [*io-list*]

Formatted: List-Directed

WRITE (*eunit, ** [,*iostat*] [,*err*]) [*io-list*]

Formatted: Namelist

WRITE (*eunit, nml-group* [,*iostat*] [,*err*])

Unformatted

WRITE (*eunit* [,*iostat*] [,*err*]) [*io-list*]

For more information, see WRITE in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Sequential WRITE Statements
- Rules for List-Directed Sequential WRITE Statements
- Rules for Namelist Sequential WRITE Statements
- Rules for Unformatted Sequential WRITE Statements

For More Information:

- See I/O control-list specifiers.
- See I/O lists.

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
```

For More Information:

- See WRITE.
- See Forms for Sequential WRITE Statements.

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.

The following table shows the default output formats for each intrinsic data type:

Default Formats for List-Directed Output	
Data Type	**Output Format**
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8) [1]	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8) [1]	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) [2]	1PG43.33E4
COMPLEX(4)	’(’,1PG14.7E2, ’, ’,1PG14.7E2, ’) ’
COMPLEX(8) T_floating	’(’,1PG23.15E3, ’, ’,1PG23.15E3, ’) ’
COMPLEX(8) D_floating	’(’,1PG23.16E2, ’, ’,1PG23.16E2, ’) ’
COMPLEX(8) G_floating	’(’,1PG23.15E3, ’, ’,1PG23.15E3, ’) ’
CHARACTER	A*w* [3]
[1] Alpha only. [2] VMS, U*X. [3] 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
```

The following shows another example:

```
  INTEGER       i, j
  REAL          a, b
  LOGICAL       on, off
  CHARACTER(20) c
  DATA i /123456/, j /500/, a /28.22/, b /.0015555/
  DATA on /.TRUE./, off/.FALSE./
  DATA c /'Here''s a string'/
  WRITE (*, *)  i, j
  WRITE (*, *)  a, b, on, off
  WRITE (*, *)  c
  END
```

The preceding example produces the following output:

```
     123456          500
  28.22000      1.555500E-03 T F
Here’s a string
```

For More Information:

- See Rules for Formatted Sequential WRITE Statements.
- On list-directed input, see Rules for List-Directed Sequential READ Statements.

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.

The following example declares a number of variables, which are placed in a namelist, initialized, and then written to the screen with namelist I/O:

```
       INTEGER(1) int1
       INTEGER    int2, int3, array(3)
       LOGICAL(1) log1
       LOGICAL log2, log3
       REAL    real1
       REAL(8) real2
       COMPLEX z1, z2
       CHARACTER(1)  char1
       CHARACTER(10) char2

       NAMELIST /example/ int1, int2, int3, log1, log2, log3,       &
      &          real1, real2, z1, z2, char1, char2, array

       int1     = 11
       int2     = 12
```

```
int3     = 14
log1     = .TRUE.
log2     = .TRUE.
log3     = .TRUE.
real1    = 24.0
real2    = 28.0d0
z1       = (38.0,0.0)
z2       = (316.0d0,0.0d0)
char1    = 'A'
char2    = '0123456789'
array(1) = 41
array(2) = 42
array(3) = 43
WRITE (*, example)
```

Output of the preceding example is:

```
&EXAMPLE
INT1  =   11,
INT2  =          12,
INT3  =          14,
LOG1  = T,
LOG2  = T,
LOG3  = T,
REAL1 =   24.00000    ,
REAL2 =   28.0000000000000     ,
Z1    = (38.00000,0.0000000E+00),
Z2    = (316.0000,0.0000000E+00),
CHAR1 = A,
CHAR2 = 0123456789,
ARRAY =          41,          42,          43
/
```

For More Information:

- See the NAMELIST statement.
- See Rules for Formatted Sequential WRITE Statements.
- On namelist input, see Rules for Namelist Sequential READ Statements.

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
```

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:

Formatted

WRITE (*eunit*, *format*, *rec* [,*iostat*] [,*err*]) [*io-list*]

Unformatted

WRITE (*eunit*, *rec* [,*iostat*] [,*err*]) [*io-list*]

For more information, see WRITE in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Direct-Access WRITE Statements
- Rules for Unformatted Direct-Access WRITE Statements

For More Information:

- See I/O control-list specifiers.
- See I/O lists.

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 one 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)
```

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)
```

Forms for Indexed WRITE Statements (VMS only)

Indexed **WRITE** statements transfer output data to external records that have keyed access. (The OPEN statement establishes the characteristics 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:

Formatted

WRITE (*eunit*, *format*, [,*iostat*] [,*err*]) [*io-list*]

Unformatted

WRITE (*eunit*, [,*iostat*] [,*err*]) [*io-list*]

For more information, see WRITE in the *A to Z Reference*.

This section discusses the following topics:

- Rules for Formatted Indexed READ Statements (VMS only)
- Rules for Unformatted Indexed READ Statements (VMS only)

For More Information:

- See I/O control-list specifiers.
- See I/O lists.

Rules for Formatted Indexed WRITE Statements (VMS only)

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.

Rules for Unformatted Indexed WRITE Statements (VMS only)

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
```

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*]

For more information on syntax, see WRITE in the *A to Z Reference*.

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:

- See I/O control-list specifiers.
- See I/O lists.
- On list-directed output, see Rules for List-Directed Sequential WRITE Statements.
- On using internal files, see your programmer's guide.

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.

For more information, see PRINT in the *A to Z Reference*.

REWRITE Statement

The **REWRITE** statement is a data transfer output statement that rewrites the current record.

A **REWRITE** statement can be formatted or unformatted. For more information, see REWRITE in the *A to Z Reference*.

I/O Formatting

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.

This chapter contains information on the following topics:

For More Information:

- On list-directed input, see Rules for List-Directed Sequential READ Statements; outout, see Rules for List-Directed Sequential WRITE Statements.
- On namelist input, see Rules for Namelist Sequential READ Statements; output, see Rules for Namelist Sequential WRITE Statements.

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, :, /, $, \, 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 BLANK Specifier in **OPEN** statements.)

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, $, +, 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.

The following table summarizes the edit descriptors that can be used in format specifications.

Summary of Edit Descriptors

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
BN	BN	Ignores embedded and trailing blanks in a numeric input field.
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
D	Dw.d	Transfers real values with D exponents.
E	Ew.d[Ee]	Transfers real values with E exponents.
EN	ENw.d [Ee]	Transfers real values with engineering notation.
ES	ESw.d [Ee]	Transfers real values with scientific notation.
F	Fw.d	Transfers real values with no exponent.
G	Gw.d[Ee]	Transfers values of all intrinsic types.
H	nHch [ch...]	Transfers characters following the H edit descriptor to an output record.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.

TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
Z	Zw[.m]	Transfers hexadecimal values.
$	$	Suppresses trailing carriage return during interactive I/O.
:	:	Terminates format control if there are no more items in the I/O list.
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.
'c' [1]	'c'	Transfers the character literal constant (between the delimiters) to an output record.
[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).

Variable format expressions must not appear in this kind of format specification.

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.

Examples

The following example shows a format specification:

```
        WRITE (*, 9000) int1, real1(3), char1
 9000   FORMAT (I5, 3F4.5, A16)
 !  I5, 3F5.2, A16 is the format list.
```

In the following example, the integer-variable name MYFMT refers to the **FORMAT** statement 9000, as assigned just before the **FORMAT** statement.

```
       ASSIGN 9000 TO MYFMT
```

```
9000  FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 is the format list.
      WRITE (*, MYFMT) iolist
```

The following shows a format example using a character expression:

```
      WRITE (*, '(I5, 3F5.2, A16)')iolist
! I5, 3F4.5, A16 is the format list.
```

In the following example, the format list is put into an 80-character variable called MYLIST:

```
CHARACTER(80) MYLIST
MYLIST = '(I5, 3F5.2, A16)'
WRITE (*, MYLIST) iolist
```

Consider the following two-dimensional array:

```
   1  2  3
   4  5  6
```

In this case, the elements are stored in memory in the order: 1, 4, 2, 5, 3, 6 as follows:

```
CHARACTER(6) array(3)
DATA array / '(I5', ',3F5.2', ',A16)' /
WRITE (*, array) iolist
```

In the following example, the **WRITE** statement uses the character array element array(2) as the format specifier for data transfer:

```
CHARACTER(80) array(5)
array(2) = '(I5, 3F5.2, A16)'
WRITE (*, array(2)) iolist
```

For More Information:

- See data edit descriptors.
- See control edit descriptors.
- See character string edit descriptors.
- See nested and group repeats.
- See printing of formatted records.

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*.

The following topics are discussed in this section:

- Forms for Data Edit Descriptors
- General Rules for Numeric Editing

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). The range of *w* is 1 through 2147483647 (2**31- 1) on Alpha processors; 1 through 32767 (2**15-1) on Intel processors. If omitted, the system applies default values (see Default Widths for Data Edit Descriptors).

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) on Alpha processors; 0 through 255 (2**8-1) on Intel processors.

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) on Alpha processors; 0 through 255 (2**8-1) on Intel processors.

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) on Alpha processors; 1 through 255 (2**8-1) on Intel processors.

Rules and Behavior

FORTRAN 77, Fortran 90, and Fortran 95 allow the field width to be omitted only for the **A** descriptor. However, DIGITAL 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, default 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:	**I**w[.m], **B**w[.m], **O**w[.m], and **Z**w[.m]
Real and complex:	**F**w.d, **E**w.d[Ee], **EN**w.d[Ee], **ES**w.d[Ee], **D**w.d, and **G**w.d[Ee]
Logical:	**L**w
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)
```

Examples

```
 !  This WRITE outputs three integers, each in a five-space field
 !  and four reals in pairs of F7.2 and F5.2 values.
       INTEGER(2) int1, int2, int3
       REAL(4) r1, r2, r3, r4
       DATA int1, int2, int3 /143, 62, 999/
       DATA r1, r2, r3, r4 /2458.32, 43.78, 664.55, 73.8/
       WRITE (*,9000) int1, int2, int3, r1, r2, r3, r4
 9000  FORMAT (3I5, 2(1X, F7.2, 1X, F5.2))
```

The following output is produced:

```
  143  62  999 2458.32 43.78  664.55 73.80
```

For More Information:

- See General rules for numeric editing.

- See Nested and group repeats.

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 **OPEN**ed 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 Terminating Short Fields of Input Data.

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:

- See Forms for data edit descriptors.
- On format specifications, in general, see Format Specifications.
- On compiler options, see your programmer's guide.

Integer Editing

Integer editing is controlled by the I (decimal), B (binary), O (octal), and Z (hexadecimal) data edit descriptors.

I Editing

The **I** edit descriptor transfers decimal integer values. It takes the following form:

I*w*[.*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 (the symbol ^ represents a nonprinting blank character):

```
Format    Input         Value
I4        2788          2788
I3        -26            -26
I9        ^^^^^^312      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 (the symbol ^ represents a nonprinting blank character):

```
Format     Value         Output
I3          284          284
I4         -284          -284
I4            0          ^^^0
I5          174          ^^174
I2         3244          **
I3         -473          ***
I7           29.812      An error; the decimal point is invalid
I4.0          0          ^^^^
I4.2          1          ^^01
I4.4          1          0001
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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 (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
B4	9	1001
B2	0	^0

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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 (the symbol ^ represents a nonprinting blank character):

```
Format    Input         Value
O5        32767               32767
O4        16234                1623
O3        97^           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 (the symbol ^ represents a nonprinting blank character):

```
Format     Value         Output
O6          32767        ^77777
O12        -32767        ^37777700001
O2          14261        **
O4             27        ^^33
O5           10.5        41050
O4.2            7        ^^07
O4.4            7        0007
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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 (the symbol ^ represents a nonprinting blank character):

```
Format       Value          Output
Z4            32767         7FFF
Z9           -32767         ^FFFF8001
Z2               16         10
Z4            -10.5         ****
Z3.3           2708         A94
Z6.4           2708         ^^0A94
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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 D, E, E1, +, -, and . are all treated as value 0.0.

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.
- On the scale factor, see Scale Factor Editing (P).
- On system default values for data edit descriptors , see Default Widths for Data Edit Descriptors.

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 (the symbol ^ represents a nonprinting blank character):

```
Format       Value              Output
F8.5            2.3547188       ^2.35472
F9.3         8789.7361          ^8789.736
F2.1           51.44            **
F10.4         -23.24352         ^^-23.2435
F5.2          325.013           ******
F5.2            -.2             -0.20
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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.

The following shows input using the **E** and **D** edit descriptors (the symbol ^ represents a nonprinting blank character):

```
Format        Input              Value
E9.3          734.432E3            734432.0
E12.4         ^^1022.43E             1022.43E-6
E15.3         52.3759663^^^^^          52.3759663
E12.5         210.5271D+10[1]        210.5271E10
BZ,D10.2      12345^^^^^           12345000.0D0
D10.2         ^^123.45^^               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 (the symbol ^ represents a nonprinting blank character):

```
Format       Value              Output
E11.2        475867.222           ^^^0.48E+06
E11.5        475867.222              0.47587E+06
E12.3             0.00069          ^^^0.690E
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          ^^^^^0.363D-01
D23.12         5413.87625793      ^^^^^0.541387625793D+04
D9.6              1.2             *********
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.
- On the scale factor, see Scale Factor Editing (P).

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.

The following shows input using the **EN** edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input           Value
EN11.3      ^^5.321E+00       5.32100
EN11.3      -600.00E-03       -.60000
EN12.3     ^^^3.150E-03        .00315
EN12.3     ^^^3.829E+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 significand 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 (the symbol ^ represents a nonprinting blank character):

```
Format     Value          Output
EN11.2     475867.222     ^475.87E+03
EN11.5     475867.222     ***********
EN12.3          0.00069   ^690.000E-06
EN10.3         -0.5555    **********
```

```
EN11.2          0.0      ^000.00E-03
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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.

The following shows input using the **ES** edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input           Value
ES11.3      ^^5.321E+00       5.32100
ES11.3       -6.000E-03       -.60000
ES12.3     ^^^3.150E-03        .00315
ES12.3     ^^^3.829E+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	$E+n_1n_2...n_e$	$E-n_1n_2...n_e$

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 **ES** edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Value          Output
ES11.2     473214.356     ^^^4.73E+05
ES11.5     473214.356     4.73214E+05
ES12.3          0.00069   ^^^6.900E-04
ES10.3         -0.5555    -5.555E-01
ES11.2          0.0       ^0.000E+00
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.

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.

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 the following table:

Table: Effect of Data Magnitude on G Format Conversions	
Data Magnitude	**Effective Conversion**
$0 < m < 0.1 - 0.5 \times 10^{-d-1}$	Ew.d[Ee]
m = 0	F(w - n).(d -1), n('b')
$0.1 - 0.5 \times 10^{-d-1} <= m < 1 - 0.5 \times 10^{-d}$	F(w - n).d, n('b')
$1 - 0.5 \times 10^{-d} <= m < 10 - 0.5 \times 10^{-d+1}$	F(w - n).(d -1), n('b')
$10 - 0.5 \times 10^{-d+1} <= m < 100 - 0.5 \times 10^{-d+2}$	F(w - n).(d -2), n('b')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2} <= m < 10^{d-1} - 0.5 \times 10^{-1}$	F(w - n).1, n('b')
$10^{d-1} - 0.5 \times 10^{-1} <= m < 10^{d} - 0.5$	(w - n).0, n('b')
$m >= 10^{d} - 0.5$	Ew.d[Ee]

The 'b' is a blank following the numeric data representation. For **G**w.d, n('b') is 4 blanks. For **G**w.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 (the symbol ^ represents a nonprinting blank character):

```
Value              Format    Output with G     Format    Output with F
       0.01234567   G13.6     ^0.123457E-01     F13.6     ^^^^^0.012346
      -0.12345678   G13.6     -0.123457^^^^     F13.6     ^^^^-0.123457
       1.23456789   G13.6     ^^1.23457^^^^     F13.6     ^^^^^1.234568
      12.34567890   G13.6     ^^12.3457^^^^     F13.6     ^^^^12.345679
     123.45678901   G13.6     ^^123.457^^^^     F13.6     ^^^123.456789
   -1234.56789012   G13.6     ^-1234.57^^^^     F13.6     ^-1234.567890
   12345.67890123   G13.6     ^^12345.7^^^^     F13.6     ^12345.678901
  123456.78901234   G13.6     ^^123457.^^^^     F13.6     123456.789012
-1234567.89012345   G13.6     -0.123457E+07     F13.6     ************
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.
- See the I data edit descriptor.
- See the L data edit descriptor.
- See the A data edit descriptor.
- On the scale factor, see Scale Factor Editing (P).

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: **F**w.d, **E**w.d[Ee], **D**w.d, **EN**w.d[Ee], **ES**w.d[Ee], or **G**w.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 (the symbol ^ represents a nonprinting blank character):

```
Format              Value                   Output
2F8.5               2.3547188, 3.456732     ^2.35472 ^3.45673
E9.2,'^,^',E5.3     47587.222, 56.123       ^0.48E+06^,^*****
```

For More Information:

- See Forms for data edit descriptors.
- See General rules for numeric editing.
- On complex constants, see General Rules for Complex Constants.

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 (the symbol ^ represents a nonprinting blank character):

```
Format      Value          Output
L5          .TRUE.         ^^^^T
```

```
L1          .FALSE.       F
```

For More Information:

See Forms for data edit descriptors.

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 the following table:

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[1]
INTEGER(1) or INTEGER*1	1
INTEGER(2) or INTEGER*2	2
INTEGER(4) or INTEGER*4	4

INTEGER(8) or INTEGER*8	8[1]
REAL(4) or REAL*4	4
DOUBLE PRECISION	8
REAL(8) or REAL*8	8
REAL(16) or REAL*16	16[2]
COMPLEX(4) or COMPLEX*8	8[3]
DOUBLE COMPLEX	16[3]
COMPLEX(8) or COMPLEX*16	16[3]

[1] Alpha only
[2] VMS, U*X
[3] Complex values are treated as pairs of real numbers, so complex editing requires a pair of real edit descriptors. (See Complex Editing.)

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 (the symbol ^ represents a nonprinting blank character):

```
Format      Input       Value         Data Type
A6          PAGE^#      #             CHARACTER(LEN=1)
A6          PAGE^#      E^#           CHARACTER(LEN=3)
A6          PAGE^#      PAGE^#        CHARACTER(LEN=6)
A6          PAGE^#      PAGE^#^^      CHARACTER(LEN=8)
A6          PAGE^#      #             LOGICAL(1)
A6          PAGE^#      ^#            INTEGER(2)
A6          PAGE^#      GE^#          REAL(4)
A6          PAGE^#      PAGE^#^^      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 (the symbol ^ represents a nonprinting blank

character):

```
Format      Value          Output
A5          OHMS           ^OHMS
A5          VOLTS          VOLTS
A5          AMPERES        AMPER
```

For More Information:

See Forms for data edit descriptors.

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 the following table:

Default Widths for Data Edit Descriptors

Edit Descriptor	Data Type of I/O List Item [1]	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)
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)	42 **d**: 33 **e**: 3
A [2], 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)	16
	CHARACTER*len	len

[1] INTEGER(8) and LOGICAL(8) are only available on Alpha processors. REAL(16) is only available on OpenVMS and DIGITAL UNIX systems.
[2] The default is the actual length of the corresponding I/O list item.

Terminating Short Fields of Input Data

On input, an edit descriptor such as **F**w.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 (*, ’(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 (*, ’(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.D0, or .FALSE. .

For More Information:

For details on input processing, see General Rules for Numeric Editing.

Control Edit Descriptors

A control edit descriptor either directly determines how text is displayed or affects the conversions performed by subsequent data edit descriptors.

The following topics are discussed in this section:

- Forms for Control Edit Descriptors
- Positional Editing
- Sign Editing
- Blank Editing
- Scale Factor Editing (P)
- Slash Editing (/)
- Colon Editing (:)
- Dollar Sign ($) and Backslash (\) Editing
- Character Count Editing (Q)

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**, **:**, **/**, \, **$**, and **Q**.

n
Is a number of character positions. It must be a positive default 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) on Alpha processors; 1 through 32767 (2**15-1) on Intel processors. 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:	:, /, \, $, 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:

- See Group repeat specifications.
- On format specifications, in general, see Format Specifications.

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.

For More Information:

See Forms for Control Edit Descriptors.

T Editing

The **T** edit descriptor specifies a character position in an I/O record. It takes the following form:

T*n*

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

In the following examples, the symbol ^ represents a nonprinting blank character.

Suppose a file has a record containing the value `ABC^^^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 Printing of Formatted Records.)

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.

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.

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:

n**X**

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 ('^K=',I6,5X)
```

Note that the symbol ^ represents a nonprinting blank character. 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('^')`.

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.

Examples

```
      INTEGER i
      REAL r

!     The following statements write:
!     251 +251 251 +251 251
      i = 251
      WRITE (*, 100) i, i, i, i, i
100   FORMAT (I5, SP, I5, SS, I5, SP, I5, S, I5)

!     The following statements write:
!     0.673E+4 +.673E+40.673E+4 +.673E+40.673E+4
      r = 67.3E2
      WRITE (*, 200) r, r, r, r, r
200   FORMAT (E8.3E1, 1X, SP, E8.3E1, SS, E8.3E1, 1X, SP, &
     &            E8.3E1, S, E8.3E1)
```

For More Information:

See Forms for Control Edit Descriptors.

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

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

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.

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, see the BLANK specifier in **OPEN** statements.)

For More Information:

See Forms for Control Edit Descriptors.

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.

Examples

If an input field formatted as a six-digit integer (I6) contains ’2 3 4’, it is interpreted as ’ 234’.

Consider the following code:

```
       READ (*, 100) n
100    FORMAT (BN, I6)
```

If you enter any one of the following three records and terminate by pressing Enter, the **READ** statement interprets that record as the value 123:

```
   123
123
123   456
```

Because the repeatable edit descriptor associated with the I/O list item n is I6, only the first six characters of each record are read (three blanks followed by 123 for the first record, and 123 followed by three blanks for the last two records). Because blanks are ignored, all three records are interpreted

as 123.

The following example shows the effect of **BN** editing with an input record that has fewer characters than the number of characters specified by the edit descriptors and *iolist*. Suppose you enter 123 and press Enter in response to the following **READ** statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. With **BN** editing in effect, the nonblank characters (123) are right-aligned, so the record is equal to 123.

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

Examples

The input field ' 23 4 ' would be interpreted as ' 23040'. If ' 23 4' were entered, the formatter would add one blank to pad the input to the six-digit integer format (I6), but this extra space would be ignored, and the input would be interpreted as ' 2304 '. The blanks following the **E** or **D** in real-number input are ignored, regardless of the form of blank interpretation in effect.

Suppose you enter 123 and press Enter in response to the following **READ** statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. If **BZ** editing is in effect, those three blanks are interpreted as zeros, and the record is equal to 123000.

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:

k**P**

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 0**P**.

Format reversion does not affect the scale factor. (For more information on format reversion, see Interaction Between Format Specifications and I/O Lists.)

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 2**P** scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2**P** 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 (the symbol ^ represents a nonprinting blank character):

```
Format    Input          Value
3PE10.5   ^^^37.614^          .037614
3PE10.5   ^^37.614E2     3761.4
-3PE10.5  ^^^^37.614    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 (the symbol ^ represents a nonprinting blank character):

```
Format     Value         Output
1PE12.3    -270.139      ^^-2.701E+02
1P,E12.2   -270.139      ^^^-2.70E+02
-1PE12.2   -270.139      ^^^-0.03E+04
```

Examples

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
```

The following code uses scale-factor editing when reading:

```
       READ (*, 100) a, b, c, d
 100   FORMAT (F10.6, 1P, F10.6, F10.6, -2P, F10.6)

       WRITE (*, 200) a, b, c, d
 200   FORMAT (4F11.3)
```

If the following data is entered:

```
 12340000  12340000  12340000  12340000
    12.34     12.34     12.34     12.34
  12.34e0   12.34e0   12.34e0   12.34e0
  12.34e3   12.34e3   12.34e3   12.34e3
```

The program's output is:

```
    12.340      1.234      1.234   1234.000
    12.340      1.234      1.234   1234.000
    12.340     12.340     12.340     12.340
 12340.000  12340.000  12340.000  12340.000
```

The next code shows scale-factor editing when writing:

```
       a = 12.34

       WRITE (*, 100) a, a, a, a, a, a
 100   FORMAT (1X, F9.4, E11.4E2, 1P, F9.4, E11.4E2, &
      &        -2P, F9.4, E11.4E2)
```

This program's output is:

```
 12.3400 0.1234E+02 123.4000 1.2340E+01   0.1234 0.0012E+04
```

For More Information:

See Forms for Control Edit Descriptors.

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) on Alpha processors; 1 through 32767 (2**15-1) on Intel processors. 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 Printing of Formatted Records).

Examples

```
!     The following statements write spreadsheet column and row labels:
      WRITE (*, 100)
100   FORMAT (’   A     B     C     D     E’                              &
     &        /,’ 1’,/,’ 2’,/,’ 3’,/,’ 4’,/,’ 5’)
```

This example generates the following output:

```
  A     B     C     D     E
1
2
3
4
5
```

For More Information:

See Forms for Control Edit Descriptors.

Colon Editing (:)

The colon edit descriptor terminates format control if no more items are in the I/O list.

Examples

Suppose the following statement 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 (the symbol ^ represents a nonprinting blank character):

```
I=^3^J=
K=13
```

```
!     The following example writes a= 3.20 b= .99
      REAL a, b, c, d
      DATA a /3.2/, b /.9871515/
      WRITE (*, 100) a, b
100   FORMAT (' a=', F5.2, :, ' b=', F5.2, :, &
     &        ' c=', F5.2, :, ' d=', F5.2)
      END
```

For More Information:

See Forms for Control Edit Descriptors.

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.

For More Information:

See Forms for Control Edit Descriptors.

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.

Examples

Consider the following:

```
       CHARACTER ICHAR(80)
       READ (4, 1000) XRAY, K, NCHAR, (ICHAR(I), I= 1, NCHAR)
 1000  FORMAT (E15.7, I4, Q, 80A1)
```

The preceding input statement reads the variables XRAY and K. The number of characters remaining in the record is NCHAR, specified by the **Q** edit descriptor. The array ICHAR is then filled by reading exactly the number of characters left in the record. (Note that this instruction will fail if NCHAR is greater than 80, the length of the array ICHAR.) By placing **Q** in the format specification, you can determine the actual length of an input record.

Note that the length returned by **Q** is the number of characters left in the record, not the number of reals or integers or other data types. The length returned by **Q** can be used immediately after it is read and can be used later in the same format statement or in a variable format expression. (See Variable Format Expressions.)

Assume the file Q.DAT contains:

```
1234.567Hello, Q Edit
```

The following program reads in the number REAL1, determines the characters left in the record, and reads those into STR:

```
       CHARACTER STR(80)
       INTEGER LENGTH
       REAL REAL1
       OPEN (UNIT = 10, FILE = 'Q.DAT')
       READ (10, 100) REAL1, LENGTH, (STR(I), I=1, LENGTH)
 100   FORMAT (F8.3, Q, 80A1)
       WRITE(*,'(F8.3,2X,I2,2X,<LENGTH>A1)') REAL1, LENGTH, (STR(I), &
      & I= 1, LENGTH)
       END
```

The output on the screen is:

```
1234.567  13  Hello, Q Edit
```

A **READ** statement that contains only a **Q** edit descriptor advances the file to the next record. For example, consider that Q.DAT contains the following data:

```
abcdefg
abcd
```

Consider it is then **READ** with the following statements:

```
       OPEN (10, FILE = "Q.DAT")
       READ(10, 100) LENGTH
 100   FORMAT(Q)
       WRITE(*,'(I2)') LENGTH
       READ(10, 100) LENGTH
       WRITE(*,'(I2)') LENGTH
       END
```

The output to the screen would be:

```
7
4
```

For More Information:

See Forms for Control Edit Descriptors.

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 Nested and Group Repeat Specifications).

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''S^DATE^IS:^',I2,'/',I2,'/',I2)
```

Note that the symbol ^ represents a nonprinting blank character.

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.

On input, the character constant edit descriptor transfers length of string characters to the edit descriptor.

Examples

Consider the following '(3I5)' format in the **WRITE** statement:

```
WRITE (10, '(3I5)') I1, I2, I3
```

This is equivalent to:

```
      WRITE (10, 100) I1, I2, I3
100   FORMAT( 3I5)
```

The following shows another example:

```
!     These WRITE statements both output ABC'DEF
!     (The leading blank is a carriage-control character).
      WRITE (*, 970)
970   FORMAT (' ABC''DEF')
      WRITE (*, '('' ABC''''DEF'')')
!     The following WRITE also outputs ABC'DEF. No carriage-
!     control character is necessary for list-directed I/O.
      WRITE (*,*) 'ABC''DEF'
```

Alternatively, if the delimiter is quotation marks, the apostrophe in the character constant `ABC'DEF` requires no special treatment:

```
WRITE (*,*) "ABC'DEF"
```

For More Information:

- See Character constants.
- On format specifications, in general, see Format Specifications.

H Editing

The **H** edit descriptor transfers data between the external record and the **H** edit descriptor itself. The **H** edit descriptor is an obsolescent Fortran 90 feature, which has been deleted in Fortran 95. DIGITAL Fortran fully supports features deleted in Fortran 95.

An **H** edit descriptor has the form of a Hollerith constant, as follows:

*n***H***string*

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) on Alpha processors; 1 through 32767 (2**15-1) on Intel processors. 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.

Examples

```
!     These WRITE statements both print "Don't misspell 'Hollerith'"
!     (The leading blanks are carriage-control characters).
!     Hollerith formatting does not require you to embed additional
!     single quotation marks as shown in the second example.
!
      WRITE (*, 960)
960   FORMAT (27H Don't misspell 'Hollerith')
      WRITE (*, 961)
961   FORMAT (' Don''t misspell ''Hollerith''')
```

For More Information:

- See Obsolescent and Deleted Language Features.
- On format specifications, in general, see Format Specifications.

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 Forms for Data Edit Descriptors.
- On group repeat specifications and format reversion, see Interaction Between Format Specifications and I/O Lists.

Variable Format Expressions

A variable format expression is a numeric expression enclosed in angle brackets (< >) that can be used in a **FORMAT** statement.

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
```

The following shows another example:

```
      WRITE(6,20) INT1
20    FORMAT(I<MAX(20,5)>)

      WRITE(6,FMT=30) REAL2(10), REAL3
30    FORMAT(<J+K>X, <2*M>F8.3)
```

The value of the expression is reevaluated each time an input/output item is processed during the execution of the **READ**, **WRITE**, or **PRINT** statement. For example:

```
      INTEGER width, value
      width=2
      READ (*,10) width, value
10    FORMAT(I1, I <width>)
      PRINT *, value
      END
```

When given input 3123, the program will print 123 and not 12.

For More Information:

For details on the synchronization of I/O lists with formats, see Interaction Between Format Specifications and I/O Lists.

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.

The following table lists the valid control characters for printing:

Control Characters for Printing

Character	Meaning	Effect
+	Overprinting	Outputs the record (at the current position in the current line) and a carriage return.
-	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.

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**, **$**, 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 (**P**), 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
```

The following example shows how several different format specifications interact with I/O lists to process data in file FOR002.DAT:

Interaction Between Format Specifications and I/O Lists

```
      INTEGER I, J, A(2,5), B(2)

      OPEN (unit=2, access='sequential', file='FOR002.DAT')

      READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2)  1
100   FORMAT (2 (I3, X, 5(I4,X), /) )  2

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)  3
999   FORMAT (' B is ', 2(I3, X), ';  A is', /
     1       (' ', 5 (I4, X)) )

      READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2)  4
200   FORMAT (2 (I3, X, 5(I4,X), :/) )

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)  5

      READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2)  6
300   FORMAT ( (I3, X, 5(I4,X)) )

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)  7

      READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2)  8
400   FORMAT ( I3, X, 5(I4,X) )
```

```
        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)   9
        END
```

1 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.

2 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.

3 This statement produces the following output:

```
B is   1   2 ;  A is
 101  102  103  104  105
 201  202  203  204  205
```

4 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.

5 This statement produces the following output:

```
B is   4   5 ;  A is
 401  402  403  404  405
 501  502  503  504  505
```

6 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.

7 This statement produces the following output:

```
B is   6   7 ;  A is
 601  602  603  604  605
 701  702  703  704  705
```

8 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.

9 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:

- See Data edit descriptors.
- See Control edit descriptors.
- See the Q edit descriptor.
- See Character string edit descriptors.
- On the scale factor, see Scale Factor Editing (P).

File Operation I/O Statements (WNT, W95, U*X)

The following are file connection, inquiry, and positioning I/O statements on Windows NT, Windows 95, and DIGITAL UNIX Systems:

- BACKSPACE

  Positions a sequential file at the beginning of the preceding record.

- CLOSE

  Terminates the connection between a logical unit and a file or device.

- DELETE

  Deletes a record from a relative file.

- ENDFILE

  Writes an end-of-file record to a sequential file and positions the file after this record.

- INQUIRE

  Requests information on the status of specified properties of a file or logical unit.

- OPEN

  Connects a Fortran logical unit to a file or device; declares attributes for read and write operations.

- REWIND

  Positions a sequential file at the beginning of the file.

- UNLOCK

  Frees a record in a relative or sequential file that was locked by a previous READ statement.

The following table summarizes I/O statement specifiers:

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
ACCESS=*access*	’SEQUENTIAL’, ’DIRECT’, or ’APPEND’	Specifies the method of file access.	INQUIRE, OPEN

ACTION=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Specifies file I/O mode.	INQUIRE, OPEN
ADVANCE=*ad_switch*	'NO' or 'YES' (default is 'YES')	Specifies formatted sequential data input as advancing, or non-advancing.	READ
ASSOCIATEVARIABLE=*var*	Integer variable	Specifies a variable to be updated to reflect the record number of the next sequential record in the file.	OPEN
BINARY=*bin*	'NO' or 'YES'	Returns whether file format is binary.	INQUIRE
BLANK=*blank_control*	'NULL' or 'ZERO' (default is 'NULL')	Specifies whether blanks are ignored in numeric fields or interpreted as zeros.	INQUIRE, OPEN
BLOCKSIZE=*blocksize*	Positive integer variable or expression	Specifies or returns the internal buffer size used in I/O.	INQUIRE, OPEN
BUFFERCOUNT=*bc*	Numeric expression	Specifies the number of buffers to be associated with the unit for multibuffered I/O.	OPEN
BUFFERED=*bf*	'YES' or 'NO' (default is 'NO')	Specifies run-time library behavior following **WRITE** operations.	INQUIRE, OPEN
CARRIAGECONTROL=*control*	'FORTRAN', 'LIST', or 'NONE'	Specifies carriage control processing.	INQUIRE, OPEN
CONVERT=*form*	'LITTLE_ENDIAN', 'BIG_ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG', or 'NATIVE' (default is 'NATIVE')	Specifies a numeric format for unformatted data.	INQUIRE, OPEN
DEFAULTFILE=*var*	Character expression	Specifies a default file pathname string.	INQUIRE, OPEN
DELIM=*delimiter*	'APOSTROPHE',	Specifies the	INQUIRE,

	’QUOTE’ or ’NONE’ (default is ’NONE’)	delimiting character for list-directed or namelist data.	OPEN
DIRECT=*dir*	’NO’ or ’YES’	Returns whether file is connected for direct access.	INQUIRE
DISPOSE=*dis* (or DISP=*dis*)	’KEEP’, ’SAVE’, ’DELETE’, ’PRINT’, ’PRINT/DELETE’, ’SUBMIT’, or ’SUBMIT/DELETE’ (default is ’DELETE’ for scratch files; ’KEEP’ for all other files)	Specifies the status of a file after the unit is closed.	OPEN, CLOSE
formatlist	Character variable or expression	Lists edit descriptors. Used in **FORMAT** statements and format specifiers (the FMT=*formatspec* option) to describe the format of data.	FORMAT, PRINT, READ, WRITE
END=*endlabel*	Integer between 1 and 99999	When an end of file is encountered, transfers control to the statement whose label is specified.	READ
EOR=*eorlabel*	Integer between 1 and 99999	When an end of record is encountered, transfers to the statement whose label is specified.	READ
ERR=*errlabel*	Integer between 1 and 99999	Specifies the label of an executable statement where execution is transferred after an I/O error.	All except **PRINT**
EXIST=*ex*	.TRUE. or .FALSE.	Returns whether a file exists and can be opened.	INQUIRE
FILE=*file* (or NAME=*name*)	Character variable or	Specifies the name of a	INQUIRE,

	expression. Length and format of the name are determined by the operating system	file	OPEN
[FMT=]*formatspec*	Character variable or expression	Specifies an *editlist* to use to format data.	PRINT, READ, WRITE
FORM=*form*	'FORMATTED', 'UNFORMATTED', or 'BINARY'	Specifies a file's format.	INQUIRE, OPEN
FORMATTED=*fmt*	'NO' or 'YES'	Returns whether a file is connected for formatted data transfer.	INQUIRE
IOFOCUS=*iof*	.TRUE. or .FALSE. (default is .TRUE. unless unit '*' is specified)	Specifies whether a unit is the active window in a QuickWin application.	INQUIRE, OPEN
iolist	List of variables of any type, character expression, or NAMELIST	Specifies items to be input or output.	PRINT, READ, WRITE
IOSTAT=*iostat*	Integer variable	Specifies a variable whose value indicates whether an I/O error has occurred.	All except **PRINT**
MAXREC=*var*	Numeric expression	Specifies the maximum number of records that can be transferred to or from a direct access file.	OPEN
MODE=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Same as ACTION.	**INQUIRE**, **OPEN**
NAMED=*var*	.TRUE. or .FALSE.	Returns whether a file is named.	INQUIRE
NEXTREC=*nr*	Integer variable	Returns where the next record can be read or written in a file.	INQUIRE
[NML=]*nmlspec*	Namelist name	Specifies a *namelist*	PRINT,

		group to be input or output.	READ, WRITE
NUMBER=n*um*	Integer variable	Returns the number of the unit connected to a file.	INQUIRE
OPENED=*od*	.TRUE. or .FALSE.	Returns whether a file is connected.	INQUIRE
ORGANIZATION=*org*	'SEQUENTIAL' or 'RELATIVE' (default is 'SEQUENTIAL')	Specifies the internal organization of a file.	INQUIRE, OPEN
PAD=*pad_switch*	'YES' or 'NO' (default is 'YES')	Specifies whether an input record is padded with blanks when the input list or format requires more data than the record holds, or whether the input record is required to contain the data indicated.	INQUIRE, OPEN
POSITION=*file_pos*	'ASIS', 'REWIND' or 'APPEND' (default is 'ASIS')	Specifies position in a file.	INQUIRE, OPEN
READ=*rd*	'NO' or 'YES'	Returns whether a file can be read.	INQUIRE
READONLY		Specifies that only READ statements can refer to this connection.	OPEN
READWRITE=*rdwr*	'NO' or 'YES'	Returns whether a file can be both read and written to.	INQUIRE
REC=*rec*	Positive integer variable or expression	Specifies the first (or only) record of a file to be read from, or written to.	READ, WRITE
RECL=*length* (or RECORDSIZE=*length*)	Positive integer variable or expression	Specifies the record length in direct access files, or the maximum record length in sequential files.	INQUIRE, OPEN

RECORDTYPE=*typ*	'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_LF', or 'STREAM_CR'	Specifies the type of records in a file.	INQUIRE, OPEN
SEQUENTIAL=*seq*	'NO' or 'YES'	Returns whether file is connected for sequential access.	INQUIRE
SHARE=*share*	'COMPAT', 'DENYNONE', 'DENYWR', 'DENYRD', or 'DENYRW' (default is 'DENYNONE')	Controls how other processes can simultaneously access a file on networked systems.	INQUIRE, OPEN
SHARED		Specifies that a file is connected for shared access by more than one program executing simultaneously.	OPEN
SIZE=*size*	Integer variable	Returns the number of characters read in a nonadvancing **READ** before an end-of-record condition occurred.	READ
STATUS=*status*	'OLD', 'NEW', 'UNKNOWN' or 'SCRATCH' (default is 'UNKNOWN')	Specifies the status of a file on opening and/or closing.	CLOSE, OPEN
TITLE=*name*	Character expression	Specifies the name of a child window in a QuickWin application.	OPEN
UNFORMATTED=*unf*	'NO' or 'YES'	Returns whether a file is connected for unformatted data transfer.	INQUIRE
[UNIT=]*unitspec*	Integer variable or expression	Specifies the unit to which a file is connected.	All except **PRINT**
USEROPEN=*fname*	Name of a user-written function	Specifies an external function that controls the opening of a file.	OPEN

WRITE=*rd*	'NO' or 'YES'	Returns whether a file can be written to.	INQUIRE

For More Information:

- See Data transfer I/O statements.
- On control specifiers, see I/O Control List.
- On record position, advancement, and transfer, see your programmer's guide.

BACKSPACE Statement

The **BACKSPACE** statement positions a file at the beginning of the preceding record, making it available for subsequent I/O processing. For more information, see BACKSPACE in the *A to Z Reference*.

CLOSE Statement

The **CLOSE** statement disconnects a file from a unit. For more information, see CLOSE in the *A to Z Reference*.

DELETE Statement

The **DELETE** statement deletes a record from a relative file. For more information, see DELETE in the *A to Z Reference*.

ENDFILE Statement

The **ENDFILE** statement writes an end-of-file record to a sequential file and positions the file after this record (the terminal point). For more information, see ENDFILE in the *A to Z Reference*.

INQUIRE Statement

The **INQUIRE** statement returns information on the status of specified properties of a file or logical unit. For more information, see INQUIRE in the *A to Z Reference*.

The following are inquiry specifiers:

- ACCESS
- ACTION
- BINARY
- BLANK
- BLOCKSIZE
- CARRIAGECONTROL
- CONVERT
- DELIM

- DIRECT
- EXIST
- FORM
- FORMATTED
- IOFOCUS
- MODE
- NAME
- NAMED
- NEXTREC
- NUMBER
- OPENED
- ORGANIZATION
- PAD
- POSITION
- READ
- READWRITE
- RECL
- RECORDTYPE
- SEQUENTIAL
- SHARE
- UNFORMATTED
- WRITE

For More Information:

- See the UNIT control specifier.
- See the ERR control specifier.
- See the IOSTAT control specifier.
- See the RECL specifier in **OPEN** statements.
- See the FILE specifier in **OPEN** statements.
- See the DEFAULTFILE specifier in **OPEN** statements.

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
'UNDEFINED'	If the file is not connected

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

BINARY Specifier (WNT, W95)

The **BINARY** specifier asks whether a file is connected to a binary file. It takes the following form:

BINARY = *bin*

bin
Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected to a binary file
'NO'	If the file is connected to a nonbinary file
'UNKNOWN'	If the file is not connected

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

BLOCKSIZE Specifier (WNT, W95)

The **BLOCKSIZE** specifier asks about the I/O buffer size. It takes the following form:

BLOCKSIZE = *bks*

bks
Is a scalar default 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.

BUFFERED Specifier

The **BUFFERED** specifier asks whether 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:

’NO’	If the file or unit is connected and buffering is not in effect.
'YES'	If the file or unit is connected and buffering is in effect.
'UNKNOWN'	If the file or unit is not connected.

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

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 DIGITAL VAX™ F_floating, D_floating, and IEEE X_floating data conversion in effect
'FGX'	If the file is connected with little endian integer and DIGITAL 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 DIGITAL VAX F_floating, D_floating, and H_floating in effect
'VAXG'	If the file is connected with little endian integer and DIGITAL 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

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

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

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.

FORM Specifier

The **FORM** specifier asks whether a file is connected for binary (WNT, W95), 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
'BINARY'	If the file is connected for binary data transfer
'UNDEFINED'	If the file is not connected

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

IOFOCUS Specifier (WNT, W95)

The **IOFOCUS** specifier asks if the indicated unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof
Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified unit is the active window in a QuickWin application
.FALSE.	If the specified unit is not the active window in a QuickWin application

If you use this specifier with a non-Windows application, an error occurs.

MODE Specifier (WNT, W95)

MODE is a nonstandard synonym for ACTION.

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. 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:

For details on the maximum size of file pathnames, see the appropriate manual in your operating system documentation set.

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

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 default 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.

NUMBER Specifier

The **NUMBER** specifier asks the number of the unit connected to a file. It takes the following form:

NUMBER = *num*

num
Is a scalar default 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, the value assigned is -1.

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

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
'UNKNOWN'	If the processor cannot determine the file's organization

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'

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

For More Information:

For details on record position, advancement, and transfer, see your programmer's guide.

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

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

RECL Specifier

The **RECL** specifier asks the maximum record length for a file. It takes the following form:

RECL = *rcl*

rcl
Is a scalar default 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 does not exist, or is not connected, the value assigned is zero.

The assigned value is expressed in 4-byte units if the file is currently (or was previously) connected for unformatted data transfer; otherwise, the value is expressed in bytes.

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 not terminated
'STREAM_CR'	If the file's records are terminated with a carriage return
'STREAM_LF'	If the file's records are terminated with a line feed
'UNKNOWN'	If the file is not connected

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

SHARE Specifier (WNT, W95)

The **SHARE** specifier asks the current share status of a file or unit. It takes the following form:

SHARE = *shr*

shr
Is a scalar default character variable that is assigned one of the following values:

’DENYRW’	If the file is connected for deny-read/write mode
'DENYWR'	If the file is connected for deny-write mode
'DENYRD'	If the file is connected for deny-read mode
'DENYNONE'	If the file is connected for deny-none mode
'UNKNOWN'	If the file or unit is not connected

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

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

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. For more information, see OPEN in the *A to Z Reference*.

The following table summarizes the **OPEN** statement specifiers:

OPEN Statement Specifiers and Values on Windows NT and Windows 95 Systems

Specifier	Values	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'	Access mode	'SEQUENTIAL'
ACTION (or MODE [1])	'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	Filesystem default
BUFFERCOUNT	n_expr	Number of I/O buffers	One
BUFFERED	'YES' 'NO'	Buffering for WRITE operations	'NO'
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	Formatted: 'LIST' [2] Unformatted: 'NONE'
CONVERT	'LITTLE_ENDIAN'	Numeric	'NATIVE'

	’BIG_ENDIAN’ ’CRAY’ ’FDX’ ’FGX’ ’IBM’ ’VAXD’ ’VAXG’ ’NATIVE’	format specification	
DEFAULTFILE	c_expr	Default file pathname	Current working directory
DELIM	'APOSTROPHE' 'QUOTE' 'NONE'	Delimiter for character constants	'NONE'
DISPOSE (or DISP)	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'	File disposition at close	'KEEP'
ERR	label	Error transfer control	No default
FILE (or NAME)	c_expr	File pathname (file name)	fort.n [3]
FORM	'FORMATTED' 'UNFORMATTED' 'BINARY' [1]	Format type	Depends on **ACCESS** setting
IOFOCUS [1]	.TRUE. or .FALSE.	Active window in QuickWin application	.TRUE. [4]
IOSTAT	var	I/O status	No default
MAXREC	n_expr	Direct access record limit	No limit
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'	File organization	'SEQUENTIAL'
PAD	'YES' 'NO'	Record padding	'YES'
POSITION	'ASIS'	File	'ASIS'

	’REWIND’ 'APPEND'	positioning	
READONLY	No value	Write protection	No default
RECL (or RECORDSIZE)	n_expr	Record length	Depends on **RECORDTYPE**, **ORGANIZATION**, and **FORM** settings [5]
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'	Record type	Depends on **ORGANIZATION**, **CARRIAGECONTROL**, **ACCESS**, and **FORM** settings
SHARE [1, 6]	'DENYRW' 'DENYWR' 'DENYRD' 'DENYNONE'	File locking	'DENYNONE'
SHARED	No value	File sharing allowed	SHARED [6]
STATUS (or TYPE)	'OLD' 'NEW' 'SCRATCH' 'REPLACE' 'UNKNOWN'	File status at open	'UNKNOWN' [7]
TITLE [1]	c_expr	Title for child window in QuickWin application	No default
UNIT	n_expr	Logical unit number	No default; an io-unit must be specified
USEROPEN	func	User program option	No default

[1] WNT, W95
[2] If you use the compiler option specifying OpenVMS defaults, and the unit is connected to a terminal, the default is ’FORTRAN’.
[3] n is the unit number.
[4] If you specify unit ’*’ the default is .FALSE..
[5] On DIGITAL UNIX systems, the default depends only on the **FORM** setting.
[6] For information on file sharing, see your user manual or programmer’s guide.
[7] The default differs under certain conditions (see STATUS Specifier).

Key to Values

```
c_expr: A scalar default character expression
func:   An external function
label:  A statement label
n_expr: A scalar numeric expression
var:    A scalar default integer variable
```

For More Information:

- On Fortran I/O status, see IOSTAT values in your programmer's guide.
- On using the **INQUIRE** statement to get file attributes of existing files, see INQUIRE Statement.
- On **OPEN** statements and file connection, see your programmer's guide.

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.
'APPEND'	Indicates sequential access, but the file is positioned at the end-of-file record.

The default is 'SEQUENTIAL'.

There are limitations on record access by file organization and record type.

For More Information:

For details on limitations on record access, see your programmer's guide.

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'.

However, if /fpscomp:general is specified on the command line and *action* is omitted, the system first attempts to open the file with 'READWRITE'. If this fails, the system tries to open the file again, first using 'READ', then using 'WRITE'.

Note that in this case, omitting *action* is not the same as specifying ACTION=’READWRITE'. If you specify ACTION='READWRITE' and the file cannot be opened for both read and write access, the attempt to open the file fails. You can use the **INQUIRE** statement to determine the actual access mode selected.

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 default integer variable. It cannot be a dummy argument to the routine in which the **OPEN** statement appears.

Direct access **READ**s, direct access **WRITE**s, 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.

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 **OPEN**ed files, preconnected files, and internal files).

If you specify /f66 (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:

For details on the **BN** and **BZ** edit descriptors, see Blank Editing.

BLOCKSIZE Specifier

The **BLOCKSIZE** specifier indicates 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.

If you specify a nonzero number for *bks*, it is rounded up to a multiple of 512 byte blocks.

If you omit BLOCKSIZE or specify zero for *bks*, the filesystem default is assumed.

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 bytes.

If you do not specify BUFFERCOUNT or you specify zero for bc, the default is 1.

For More Information:

- See the BLOCKSIZE specifier.
- On obtaining optimal run-time performance, see your programmer's guide.

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'.

If BUFFERED='YES' is specified, the request may or may not be honored, depending on the output device and other file or connection characteristics.

If BLOCKSIZE and BUFFERCOUNT have been specified for **OPEN**, their product determines the size in bytes of the internal buffer. Otherwise, the default size of the internal buffer is 8192 bytes.

Note: The default size of the internal buffer is 1024 bytes if compiler option /fpscomp=general is used.

The internal buffer will grow to hold the largest single record but will never shrink.

CARRIAGECONTROL Specifier

The **CARRIAGECONTROL** specifier indicates the type of carriage control used when a file is displayed at a terminal. 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 binary (WNT, W95) and unformatted files is 'NONE'. The default for formatted files is 'LIST'. However, if you specify /vms or /fpscomp=general, and the unit is connected to a terminal, the default is 'FORTRAN'.

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect or the file was 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 is used to control vertical spacing.

For More Information:

For details on valid control characters for printing, see Printing of Formatted Records.

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 DIGITAL 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) [4].
'FGX'	Little endian integer data [2] and DIGITAL 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)

[4].

'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 DIGITAL 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) [4].
'VAXG'	Little endian integer data [2] and DIGITAL 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) [4].
'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), or COMPLEX(8)
[4] U*X only

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 environment variable, the compiler option /convert, or OPTIONS/CONVERT. The following shows the order of precedence:

Method Used	**Precedence**
An environment variable	Highest (1)
OPEN(CONVERT=*convert*)	2
OPTIONS/CONVERT	3
The /convert:k*eyword* compiler option	Lowest (4)

The /convert compiler option and **OPTIONS**/CONVERT affect all unit numbers used by the program, while environment variables 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:

- See Environment Variables Used with the DF Command
- See Run-Time Environment Variables
- On supported ranges for data types, see Data Types, Constants, and Variables and your programmer's guide.
- On compiler options, in general, see your programmer's guide.

DEFAULTFILE Specifier

The **DEFAULTFILE** specifier indicates a default file pathname string. It takes the following form:

DEFAULTFILE = *def*

def
Is a character expression indicating a default file pathname string.

The default file pathname string is used primarily when accepting file pathnames interactively. File pathnames known to a user program normally appear in the FILE specifier.

DEFAULTFILE supplies a value to the Fortran I/O system that is prefixed to the name that appears in FILE.

If *def* does not end in a slash (/), a slash is added.

If DEFAULTFILE is omitted, the Fortran I/O system uses the current working directory.

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.

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 line printer spooler and retains it.
'PRINT/DELETE' [1]	Submits the file to the line printer spooler and then deletes it.
'SUBMIT'	Forks a process to execute the file.
'SUBMIT/DELETE'	Forks a process to execute the file, and then deletes the file after the fork is completed.

[1] Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

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 pathname allowed by the operating system.

Any trailing blanks in the name are ignored.

If FILE is omitted and the unit is not connected to a file, the **OPEN** statement must specify STATUS='SCRATCH'.

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.

If the filename is 'USER' or 'CON', input and output are directed to the console. For a complete list of device names, see Physical Devices.

In a QuickWin application, you can specify FILE='USER' to open a child window. All subsequent I/O statements directed to that unit appear in the child window.

The *name* can be blank (FILE=' ') if the compatibility compiler option /fpscomp:filesfromcmd is specified. If the *name* is blank, the following occurs:

1. The program reads a filename from the list of arguments (if any) in the command line that started the program. If the argument is a null or blank string (" "), you are prompted for the corresponding filename. Each successive **OPEN** statement that specifies a blank name reads the next following command-line argument.

2. If no command-line arguments are specified or there are no more arguments in the list, you are prompted for additional filenames.
   Assume the following command line started the program MYPROG (note that quotation marks (") are used):

   myprog first.fil " " third.txt

   MYPROG contains four **OPEN** statements with blank filenames, in the following order:

   ```
   OPEN (2, FILE = ' ')
   OPEN (4, FILE = ' ')
   OPEN (5, FILE = ' ')
   OPEN (10, FILE = ' ')
   ```

   Unit 2 is associated with the file FIRST.FIL. Because a blank argument was specified on the command line for the second filename, the **OPEN** statement for unit 4 produces the following prompt:

   ```
   Filename missing or blank -
   ```

```
Please enter name UNIT 4?
```

Unit 5 is associated with the file THIRD.TXT. Because no fourth file was specified on the command line, the **OPEN** statement for unit 10 produces the following prompt:

```
Filename missing or blank -
Please enter name UNIT 10?
```

For More Information:

- See Physical Devices in Files, Devices, and I/O Hardware in the *Programmer's Guide*.
- On default file name conventions, see your programmer's guide.
- On allowable file pathnames, see the appropriate manual in your system documentation set.

FORM Specifier

The **FORM** specifier indicates whether the file is being connected for binary (WNT, W95), 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
'BINARY'	Indicates binary data transfer

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct access files.

The data is stored and retrieved in a file according to the file's access (set by the ACCESS specifier) and the form of the data the file contains.

A *formatted file* is a sequence of formatted records. Formatted records are a series of ASCII characters terminated by an end-of-record mark (a carriage return and line feed sequence). The records in a formatted direct-access file must all be the same length. The records in a formatted sequential file can have varying lengths. All internal files must be formatted.

An *unformatted file* is a sequence of unformatted records. An unformatted record is a sequence of values. Unformatted direct files contain only this data, and each record is padded to a fixed length with undefined bytes. Unformatted sequential files contain the data plus information that indicates the boundaries of each record.

Binary sequential files are sequences of bytes with no internal structure. There are no records. The file contains only the information specified as I/O list items in **WRITE** statements referring to the file.

Binary direct files have very little structure. A record length is assigned by the RECL specifier in an **OPEN** statement. This establishes record boundaries, which are used only for repositioning and padding before and after read and write operations and during **BACKSPACE** operations. Record boundaries do not restrict the number of bytes that can be transferred during a read or write operation. If an I/O operation attempts to read or write more values than are contained in a record, the read or write operation is continued on the next record.

IOFOCUS Specifier (WNT, W95)

The **IOFOCUS** specifier indicates whether a particular unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof
Is a scalar default logical expression that evaluates to one of the following values:

.TRUE.	Indicates the QuickWin child window is the active window
.FALSE.	Indicates the QuickWin child window is not the active window

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any **READ**, **WRITE**, or **PRINT** statement to that window. **OUTTEXT**, **OUTGTEXT**, or any other graphics routine call does not cause the focus to shift.

For More Information:

See Giving a Window Focus and Setting the Active Window in Using QuickWin in the *Programmer's Guide*.

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 an unlimited number of records.

MODE Specifier (WNT, W95)

MODE is a nonstandard synonym for ACTION.

NAME Specifier

NAME is a nonstandard synonym for FILE.

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.

The default is 'SEQUENTIAL'.

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, DIGITAL 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.

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'. (On Fortran I/O systems, this is the same as 'REWIND'.)

A new file (whether specified as new explicitly or by default) is always positioned at its initial point.

In addition to the POSITION= specifier, you can use position statements. The BACKSPACE statement positions a file back one record. The REWIND statement positions a file at its initial point. The ENDFILE statement writes an end-of-file record at the current position and positions the file after it. Note that **ENDFILE** does not go the end of an existing file, but creates an end-of-file where it is.

For More Information:

For details on record position, advancement, and transfer, see your programmer's guide.

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.

The Fortran I/O system's default privileges for file access are READWRITE. If access is denied, the I/O system automatically retries accessing the file for **READ** access.

However, if you use the /vms compiler option, the I/O system does not retry accessing for **READ** access. So, run- time I/O errors can occur if the file protection does not permit **WRITE** access. To prevent such errors, if you wish to read a file for which you do not have write access, specify READONLY.

RECL Specifier

The **RECL** specifier indicates the length of each record in a file connected for direct 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 the /assume:byterecl compiler option is specified.

Except for segmented records, the *rl* is the length for record data only, it does not include space for 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 (including the four bytes of control information).

- For fixed-length records, RECL indicates the size of each record; it *must* be specified. If the records are unformatted, the size must be expressed as an even multiple of four.

  You can use the RECL specifier in an **INQUIRE** statement to get the record length before opening the file.

- For variable-length records, RECL indicates the maximum length for any record.

If you read a fixed-length file with a record length different from the one used to create the file, indeterminate results can occur.

The maximum length for *rl* depends on the record type and the setting of the CARRIAGECONTROL specifier, as shown in the following table:

Maximum Record Lengths (RECL) on DIGITAL UNIX, Windows NT, and Windows 95 Systems

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Fixed-length	'NONE'	2147483647 (2**31-1) [1]
Variable-length	'NONE'	2147483640 (2**31-8)
Segmented	'NONE'	32764 (2**15-4)
Stream	'NONE'	2147483647 (2**31-1)
Stream_CR	'LIST'	2147483647 (2**31-1)
	'FORTRAN'	2147483646 (2**31-2)
Stream_LF	'LIST'	2147483647 (2**31-1) [2]
	'FORTRAN'	2147483646 (2**31-2)

[1] Subtract 1 if the /vms compiler option is used.
[2] U*X only

The default value depends on the setting of the RECORDTYPE specifier, as shown in the following table:

Default Record Lengths (RECL) on DIGITAL UNIX, Windows NT, and Windows 95 Systems

RECORDTYPE	RECL value
'FIXED'	None; value must be explicitly specified.
All other settings	132 bytes for formatted records; 510 longwords for unformatted records.

RECORDSIZE Specifier

RECORDSIZE is a nonstandard synonym for RECL.

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_LF'	Indicates stream-type variable length records, terminated with a line feed.
'STREAM_CR'	Indicates stream-type variable length records, terminated with a carriage return.

When you open a file, default record types are as follows:

'FIXED'	For relative files
'FIXED'	For direct access sequential files
'STREAM_LF'	For formatted sequential access files
'VARIABLE'	For unformatted sequential access files

A *segmented record* is a logical record consisting of segments that are physical records. Since the length of a segmented record can be greater than 65,535 bytes, only use segmented records for unformatted sequential access to disk or raw magnetic tape files.

Files containing segmented records can be accessed only by unformatted sequential data transfer statements.

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:

For details on record types and file organization, see your programmer's guide.

SHARE Specifier (WNT, W95)

The **SHARE** specifier indicates whether file locking is implemented while the unit is open. It takes the following form:

SHARE = *shr*

shr
Is a scalar default character expression that evaluates to one of the following values:

'DENYRW'	Indicates deny-read/write mode. No other process can open the file.
'DENYWR'	Indicates deny-write mode. No process can open the file with write access.
'DENYRD'	Indicates deny-read mode. No process can open the file with read access.
'DENYNONE'	Indicates deny-none mode. Any process can open the file in any mode.

The default is 'DENYNONE'.

'COMPAT' is accepted for compatibility with previous versions. It is equivalent to 'DENYNONE'.

Use the ACCESS specifier in an **INQUIRE** statement to determine the access permission for a file.

Be careful not to permit other users to perform operations that might cause problems. For example, if you open a file intending only to read from it, and want no other user to write to it while you have it open, you could open it with ACTION='READ' and SHARE='DENYRW'. Other users would not be able to open it with ACTION='WRITE' and change the file.

Suppose you want several users to read a file, and you want to make sure no user updates the file while anyone is reading it. First, determine what type of access to the file you want to allow the original user. Because you want the initial user to read the file only, that user should open the file with ACTION='READ'. Next, determine what type of access the initial user should allow other users; in this case, other users should be able only to read the file. The first user should open the file with SHARE='DENYWR'. Other users can also open the same file with ACTION='READ' and SHARE='DENYWR'.

For More Information:

For details on limitations on record access, see your programmer's guide.

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

Shared access is the default for the Fortran I/O system if the /fpscomp:general compiler option is specified.

For More Information:

For details on file sharing, see your programmer's guide.

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 and its status changes to 'OLD'.

Scratch files go into a temporary directory and are visible while they are open. Scratch files are deleted when the unit is closed or when the program terminates normally, whichever occurs first. You can use the TMP or TEMP environment variable to specify the path for scratch files; if neither environment variable is defined, the default is the current directory.

The default is 'UNKNOWN'. This is also the default if you implicitly open a file by using **WRITE**.

However, if you implicitly open a file using **READ**, the default is 'OLD'. If you specify the /f66 compiler option (or OPTIONS/NOF77), the default is 'NEW'.

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.

TITLE Specifier

The **TITLE** specifier indicates the name of a child window in a QuickWin application. It takes the following form:

TITLE = *name*

name
Is a character expression.

If TITLE is specified in a non-Quickwin application, a run-time error occurs.

For More Information:

For details on QuickWin applications, see Using QuickWin in the *Programmer's Guide*.

TYPE Specifier

TYPE is a nonstandard synonym for STATUS.

USEROPEN Specifier

The **USEROPEN** specifier lets you pass control to a routine that directly opens a file. The file can use system calls or library routines to establish a special context that changes the effect of subsequent Fortran I/O statements.

The **USEROPEN** specifier takes the following form:

USEROPEN = *function-name*

function-name
Is the name of an external function.

The Visual Fortran Run-time Library (RTL) I/O support routines call the function named in USEROPEN in place of the system calls normally used when the file is first opened for I/O. On WIN32 platforms, the Fortran RTL would normally call CreateFile( ) to open a file.

The called function must open the file (or pipe, etc.) using CreateFile( ) and return the *handle* of the file (return value from CreateFile( )) when it returns control to the calling Visual Fortran program. When opening the file, the called function usually specifies options different from those provided by

a normal Fortran **OPEN** statement.

The main purpose of the function named in USEROPEN is to jacket a call to the CreateFile( ) WIN32 api. The function can be written in Fortran, C, or other languages. If the function is written in Fortran, do not execute a Fortran **OPEN** statement to open the file named in USEROPEN.

Examples

In the calling Fortran program, the function named in USEROPEN must first be declared in an **EXTERNAL** statement. For example, the following Fortran code might be used to call the USEROPEN procedure UOPEN:

```
IMPLICIT INTEGER (A-Z)
EXTERNAL UOPEN
...
OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='NEW',USEROPEN=UOPEN)
```

When the **OPEN** statement is executed, the UOPEN function receives control. The function opens the file by calling CreateFile( ), performs whatever operations were specified, and subsequently returns control (with the *handle* returned by CreateFile( )) to the calling Fortran program.

Here is what the UOPEN function might look like:

```
        INTEGER FUNCTION UOPEN( FILENAME,        &
                                DESIRED_ACCESS,  &
                                SHARE_MODE,      &
                                A_NULL,          &
                                CREATE_DISP,     &
                                FLAGS_ATTR,      &
                                B_NULL,          &
                                UNIT,            &
                                FLEN )
        !DEC$ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
        !DEC$ATTRIBUTES REFERENCE :: DESIRED_ACCESS
        !DEC$ATTRIBUTES REFERENCE :: SHARE_MODE
        !DEC$ATTRIBUTES REFERENCE :: CREATE_DISP
        !DEC$ATTRIBUTES REFERENCE :: FLAGS_ATTR
        !DEC$ATTRIBUTES REFERENCE :: UNIT

        USE DFWIN

        IMPLICIT INTEGER (A-Z)
        CHARACTER*(FLEN) FILENAME
        TYPE(T_SECURITY_ATTRIBUTES), POINTER :: NULL_SEC_ATTR

! Set the FILE_FLAG_WRITE_THROUGH bit in the flag attributes to CreateFile( )
! (for whatever reason)
        FLAGS_ATTR = FLAGS_ATTR + FILE_FLAG_WRITE_THROUGH

! Do the CreateFile( ) call and return the status to the Fortran rtl
        STS = CreateFile( FILENAME,              &
                          DESIRED_ACCESS,        &
                          SHARE_MODE,            &
                          NULL_SEC_ATTR,         &
                          CREATE_DISP,           &
                          FLAGS_ATTR,            &
```

```
                    0 )

      UOPEN = STS
      RETURN

      END
```

The UOPEN function is declared to use the cdecl calling convention, so it matches the Fortran rtl declaration of a useropen routine.

The following function definition and arguments are passed from the Visual Fortran Run-time Library to the function named in USEROPEN:

```
      INTEGER FUNCTION UOPEN( FILENAME,        &
                              DESIRED_ACCESS,  &
                              SHARE_MODE,      &
                              A_NULL,          &
                              CREATE_DISP,     &
                              FLAGS_ATTR,      &
                              B_NULL,          &
                              UNIT,            &
                              FLEN )
      !DEC$ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
      !DEC$ATTRIBUTES REFERENCE :: DESIRED_ACCESS
      !DEC$ATTRIBUTES REFERENCE :: SHARE_MODE
      !DEC$ATTRIBUTES REFERENCE :: CREATE_DISP
      !DEC$ATTRIBUTES REFERENCE :: FLAGS_ATTR
      !DEC$ATTRIBUTES REFERENCE :: UNIT
```

The first 7 arguments correspond to the CreateFile( ) api arguments. The value of these arguments is set according the caller's **OPEN**( ) arguments:

FILENAME
: Is the address of a null terminated character string that is the name of the file.

DESIRED_ACCESS
: Is the desired access (read-write) mode passed by reference.

SHARE_MODE
: Is the file sharing mode passed by reference.

A_NULL
: Is always null. The Fortran runtime library always passes a NULL for the pointer to a SECURITY_ATTRIBUTES structure in its CreateFile( ) call.

CREATE_DISP
: Is the creation disposition specifying what action to take on files that exist, and what action to take on files that do not exist. It is passed by reference.

FLAGS_ATTR
: Specifies the file attributes and flags for the file. It is passed by reference.

B_NULL
: Is always null. The Fortran runtime library always passes a NULL for the handle to a template file in it's CreateFile( ) call.

The last 2 arguments are the Fortran unit number and length of the file name:

UNIT

Is the Fortran unit number on which this **OPEN** is being done. It is passed by reference.

FLEN

Is the length of the file name, not counting the terminating null, and passed by value.

REWIND Statement

The **REWIND** statement positions a sequential file at the beginning of the file (the initial point). For more information, see REWIND in the *A to Z Reference*.

UNLOCK Statement

The **UNLOCK** statement frees a record in a relative or sequential file that was locked by a previous **READ** statement. For more information, see UNLOCK in the *A to Z Reference*.

Compilation Control Statements and Compiler Directives

You can specify certain statements and directives within programs to influence compilation.

This chapter contains information on the following topics:

- Compilation control statements
- General Compiler directives

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:

- The INCLUDE Statement

  Incorporates external source code into programs.

- The OPTIONS Statement

  Sets options usually specified in the compiler command line. **OPTIONS** statement settings override command line options.

General Compiler Directives

DIGITAL Fortran provides several general-purpose compiler directives to perform tasks during compilation. You do not need to specify a compiler option to enable general directives.

The following general compiler directives are available:

- ALIAS

  Specifies an alternate external name to be used when referring to external subprograms.

- ATTRIBUTES

  Specifies properties for data objects and procedures.

- DECLARE and NODECLARE

  Generates or disables warnings for variables that have been used but not declared.

- DEFINE and UNDEFINE

  Specifies a symbolic variable whose existence (or value) can be tested during conditional compilation.

- FIXEDFORMLINESIZE

  Sets the line length for fixed-form source code.

- FREEFORM and NOFREEFORM

  Specifies free-format or fixed-format source code.

- IDENT

  Specifies an identifier for an object module.

- IF and IF DEFINED

  Specifies a conditional compilation construct.

- INTEGER

  Specifies the default integer kind.

- MESSAGE

  Specifies a character string to be sent to the standard output device during the first compiler pass.

- OBJCOMMENT

  Specifies a library search path in an object file.

- OPTIONS

  Controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries.

- PACK

  Specifies the memory starting addresses of derived-type items.

- PSECT

  Modifies certain characteristics of a common block.

- REAL

  Specifies the default real kind.

- STRICT and NOSTRICT

Disables or enables language features not found in the Fortran 90 language standard.

- TITLE and SUBTITLE

  Specifies a title or subtitle for a listing header.

The following sections describe:

- Syntax Rules for General Directives
- Equivalent Compiler Options

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:

***c*DEC$**

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 or 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. A directive ends in column 72 (or column 132, if compiler option /extend_source is specified).

- Prefixes beginning with ! are allowed in all source forms.

  In fixed and tab source forms, a prefix beginning with ! must follow the same rules for prefixes beginning with C, c, or * (see above).

  In free source form, the prefix need not start in column 1, but it cannot be preceded by any nonblank characters on the same line. It can only be preceded by whitespace.

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 (/ /).

For More Information:

For more details, see General Compiler Directives.

Equivalent Compiler Options

Some compiler directives and compiler options have the same effect (see the following table). However, compiler directives can be turned on and off throughout a program, while compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

Compiler directive	Equivalent command-line compiler option
DECLARE	/warn:declarations or /4Yd
NODECLARE	/warn:nodeclarations or /4Nd
DEFINE symbol	/define:*symbol* or /D*symbol*
FIXEDFORMLINESIZE:option	/extend_source[:*option*] or /4L *option*
FREEFORM	/free or /nofixed, or /4Yf
NOFREEFORM	/nofree, /fixed, or /4Nf
INTEGER:option	/integer_size:*option* or /4I *option*
OBJCOMMENT	/libdir
PACK:option	/alignment[:*option*] or /Zp *option*
REAL:option	/real_size:*option* or /4R *option*
STRICT	/warn:stderrors with /stand:f90 or /4Ys
NOSTRICT	/4Ns

Note that any of the compiler directive names above can be specified using the prefix **!MS$**; for example, **!MS$NOSTRICT** is allowed.

For rules on using compiler directives, see Syntax Rules for General Directives.

Scope and Association

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.

Program entities are accessible within a scope that can be any of the following:

- An entire executable program
- A single scoping unit
- A 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.

This section contains information on the following topics:

Scope

Program entities have the following kinds of scope (as shown in the table below):

- Global

  Entities that are accessible throughout an executable program. The name of a global entity 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 the table below).

  A *scoping unit* is one of the following:

  - A derived-type definition
  - A procedure interface body (excluding any derived-type definitions and interface bodies contained within it)
  - A 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 Use and Host Association.)

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 the table below.

- 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.

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	
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 Defining Generic Assignment).

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:

- See Derived data types.
- On user-defined generic procedures, see Defining Generic Names for Procedures.
- See Intrinsic procedures.
- On procedures and subprograms, see Program Units and Procedures.
- See Use and host association.
- On defined operations, see Defining Generic Operators.
- On defined assignment, see Defining Generic Assignment.
- On how the PRIVATE attribute can affect accessibility of entities, see PRIVATE and PUBLIC Attributes and Statements.

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 Defining Generic Names for Procedures.

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.

This section discusses the following topics:

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.

Examples

The following example shows how a module can define three separate procedures, and a main program give them a generic name `DUP` through an interface block. Although the main program calls all three by the generic name, there is no ambiguity since the arguments are of different data types, and `DUP` is a function rather than a subroutine. The module `UN_MOD` must give each procedure a different name.

```
MODULE UN_MOD
!

CONTAINS
   subroutine dup1(x,y)
   real x,y
   print *, ’ Real arguments’, x, y
   end subroutine dup1

   subroutine dup2(m,n)
   integer m,n
   print *, ’ Integer arguments’, m, n
   end subroutine dup2

   character function dup3 (z)
   character(len=2) z
   dup3 = ’String argument ’// z
   end function dup3

END MODULE

program unclear
!
! demonstrates how to use generic procedure references

USE UN_MOD
INTERFACE DUP
   MODULE PROCEDURE dup1, dup2, dup3
END INTERFACE

real a,b
integer c,d
character (len=2) state

a = 1.5
b = 2.32
c = 5
d = 47
```

```
state = 'WA'

call dup(a,b)
call dup(c,d)
print *, dup(state)       !actual output is 'S'only
END
```

Note that the function `DUP3` only prints one character, since module `UN_MOD` specifies no length parameter for the function result.

If the dummy arguments `x` and `y` for `DUP` were declared as integers instead of reals, then any calls to `DUP` would be ambiguous. If this is the case, a compile-time error results.

The subroutine definitions, `DUP1`, `DUP2`, and `DUP3`, must have different names. The generic name is specified in the first line of the interface block, and in the example is `DUP`.

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.

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:

- See Function references.

- See the USE statement.
- On subroutine references, see the CALL Statement.
- On generic procedure names, see Defining Generic Names for Procedures.

Association

Entities are associated when each is associated with the same storage location.

Two (or more) entities can become associated by the following:

- Name association
- Pointer association
- Storage association

The following example shows name, pointer, and storage association between an external program unit and an external procedure.

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
    ...
```

Name Association

Name association allows an entity 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.

Argument Association

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 more details, see also Argument Association.

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.

Examples

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
```

The following example shows how a host and an internal procedure can use host-associated entities:

```
program INTERNAL
! shows use of internal subroutine and CONTAINS statement
   real a,b,c
   call find
   print *, c
contains
   subroutine find
     read *, a,b
     c = sqrt(a**2 + b**2)
   end subroutine find
end
```

In this example, the variables `a`, `b`, and `c` are available to the internal subroutine `find` through host association. They do not have to be passed as arguments to the internal procedure. In fact, if they are, they become local variables to the subroutine and hide the variables declared in the host program.

Conversely, the host program knows the value of `c`, when it returns from the internal subroutine that has defined `c`.

For More Information:

- See the USE statement.
- On entities with local scope, see Scope.

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:

- See Pointer assignments.
- See the NULL intrinsic function.
- On the **ALLOCATE** and **DEALLOCATE** statements, see Dynamic Allocation.
- On the **NULLIFY** statement, see Dynamic Allocation

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.

This section discusses the following topics:

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 DIGITAL 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 DIGITAL 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.

The following table lists the storage requirements (in bytes) for the intrinsic data types:

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) [2]	8
INTEGER	2, 4 or 8 [1]
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8) [2]	8

REAL	4 or 8 [3]
REAL(4)	4
DOUBLE PRECISION	8
REAL(8)	8
REAL(16) [4]	16
COMPLEX	8 or 16 [3]
COMPLEX(4)	8
DOUBLE COMPLEX	16
COMPLEX(8)	16
CHARACTER	1
CHARACTER*len	len [5]
CHARACTER*(*)	assumed-length [6]

[1] Depending on default integer, LOGICAL and INTEGER can have two, four, or eight bytes. The default allocation is four bytes.
[2] Alpha only
[3] Depending on default real, REAL can have four or eight bytes and COMPLEX can have eight or sixteen bytes. The default allocations are four bytes for REAL and eight bytes for COMPLEX.
[4] VMS, U*X
[5] The value of len is the number of characters specified. The largest valid value is 2147483647 (2**31-1) for DIGITAL UNIX, Windows NT, and Windows 95 systems; 65535 for OpenVMS systems. Negative values are treated as zero.
[6] 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 Assumed-Length Character Arguments and your programmer's guide.)

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:

- See the COMMON statement.
- See the ENTRY statement.
- See the EQUIVALENCE statement.
- On the hardware representations of data types, see your programmer's guide.

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:

- See Arrays.
- See the DATA statement.
- On array element order, see Array Elements.

Obsolescent and Deleted Language Features

Fortran 90 identifies some FORTRAN 77 features to be obsolescent. Fortran 95 deletes some of these features, and identifies a few more language features to be obsolescent. Features considered obsolescent may be removed from future revisions of the Fortran Standard.

You can specify the /standcompiler option to have these features flagged.

Note: DIGITAL Fortran fully supports features deleted from Fortran 95.

This section discusses the following topics:

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.

DIGITAL Fortran flags these features if you specify the /stand compiler option.

Fortran 90 suggests other methods 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 or CASE construct.

- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct.

- **ASSIGN** and assigned **GO TO** statements

  These statements are usually used to simulate internal procedures, 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.

- 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 IF Construct).

- **H** edit descriptor

  To replace this functionality, it is recommended that you use the character constant edit descriptor (see Character Constant Editing).

- **PAUSE** statement

  To replace this functionality, it is recommended that you use a READ statement that awaits input data.

- 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 DO Constructs).

- 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 Forms for DO Constructs) or a CONTINUE statement.

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 DIGITAL 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

DIGITAL Fortran flags these features if you specify the /stand compiler option.

For suggested methods to achieve the functionality of these features, see Obsolescent Language Features in Fortran 90.

Obsolescent Language Features in Fortran 95

Some language features, considered redundant in Fortran 90 are identified as obsolescent in Fortran 95.

DIGITAL Fortran flags these features if you specify the /stand compiler option.

Fortran 90 offers other methods 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.

- Arithmetic **IF**

  To replace this functionality, it is recommended that you use an IF statement or construct.

- 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 Declaration Statements for Character Types).

- Computed **GO TO** statement

  To replace this functionality, it is recommended that you use a CASE construct.

- **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.

- 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 Forms

for DO Constructs) or a CONTINUE statement.

- Statement functions

  To replace this functionality, it is recommended that you use an internal function.

Additional Language Features

To facilitate compatibility with older versions of Fortran, DIGITAL Fortran provides the following additional language features:

- The DEFINE FILE statement
- The ENCODE and DECODE statements
- The FIND statement
- FORTRAN-66 Interpretation of the EXTERNAL Statement
- An alternative syntax for the PARAMETER statement
- The VIRTUAL statement
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for a record specifier
- An alternate syntax for the DELETE statement
- An alternative form for namelist external records
- The DIGITAL Fortran POINTER statement
- Record structures

These language features are particularly useful in porting older Fortran programs to Fortran 90. However, you should avoid using them in new programs on these systems, and in new programs for which portability to other Fortran 90 implementations is important.

FORTRAN-66 Interpretation of the EXTERNAL Statement

If you specify compiler option /f66, the EXTERNALstatement 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 with that of the **EXTERNAL** statement.

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 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 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

The following example demonstrates the FORTRAN-66 **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 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.

Alternative Syntax for the PARAMETER Statement

The **PARAMETER** statement discussed here is similar to the one discussed in PARAMETER; 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=3.14159265358979238D0
PARAMETER PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

For More Information:

For details on compile-time constant expressions, see PARAMETER.

Alternative Syntax for Octal and Hexadecimal Constants

In DIGITAL 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'
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:

- See Octal constants.
- See Hexadecimal constants.

Alternative Syntax for a Record Specifier

In DIGITAL 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.

Alternative Syntax for the DELETE Statement

In DIGITAL 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:

See also the DELETE statement.

Alternative Form for Namelist External Records

In DIGITAL 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:

- See the NAMELIST statement.
- On namelist input, see Rules for Namelist Sequential READ Statements.
- On namelist output, see Rules for Namelist Sequential WRITE Statements.

Record Structures

The record structure was defined in earlier versions of DIGITAL Fortran as a language extension. It is still supported in Visual Fortran, although its functionality has been replaced by standard Fortran 90 derived types. Record structures in existing code can be easily converted to Fortran 90 derived type structures for portability, but can also be left in their old form. In most cases, a DIGITAL Fortran record and a Fortran 90 derived type can be used interchangeably.

DIGITAL Fortran record structures are similar to Fortran 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.)

Examples

DIGITAL Fortran record structures, using only intrinsic types, easily convert to Fortran 90 derived types. The conversion can be as simple as replacing the keyword **STRUCTURE** with **TYPE** and removing slash (/) marks. The following shows an example conversion:

Record Structure

```
STRUCTURE /employee_name/
  CHARACTER*25  last_name
  CHARACTER*15  first_name
END STRUCTURE
STRUCTURE /employee_addr/
  CHARACTER*20  street_name
  INTEGER(2)    street_number
  INTEGER(2)    apt_number
  CHARACTER*20  city
  CHARACTER*2   state
  INTEGER(4)    zip
END STRUCTURE
```

Fortran 90 Derived-Type

```
TYPE employee_name
  CHARACTER*25  last_name
  CHARACTER*15  first_name
END TYPE
TYPE employee_addr
  CHARACTER*20  street_name
  INTEGER(2)    street_number
  INTEGER(2)    apt_number
  CHARACTER*20  city
  CHARACTER*2   state
  INTEGER(4)    zip
END TYPE
```

The record strcutures can be used as subordinate record variables within another record, such as the `employee_data` record. The equivalent Fortran 90 derived type would use the derived-type objects as components in a similar manner, as shown below:

Record Structure

```
STRUCTURE /employee_data/
  RECORD  /employee_name/ name
  RECORD  /employee_addr/ addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END STRUCTURE
```

Fortran 90 Derived-Type

```
TYPE employee_data
  TYPE (employee_name) name
  TYPE (employee_addr) addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

The following topics are also related to record structures:

- Structure Declarations
- RECORD Statement
- References to Record Fields
- Aggregate Assignment

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. For more information, see STRUCTURE in the *A to Z Reference*.

The following are related topics:

- Type Declarations
- Substructure Declarations
- Union Declarations

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.

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 STRUCTURE for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

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:

Aggregate Field Reference

record-name [**.***aggregate-field-name*] ...

Scalar Field Reference

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 **EQUIVALENCE** statements. However, you can make fields of record structures equivalent to themselves by using the **UNION** and **MAP** statements in a structure declaration.

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. DIGITAL Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

DIGITAL 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, DIGITAL 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
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:

- See the RECORD statement.
- On specification of fields within structure declarations, see the STRUCTURE statement.
- On structure declarations, see the STRUCTURE statement.
- On **UNION** and **MAP** statements, see the UNION statement.
- On alignment of data, see your programmer's guide.

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
```

Character and Key Code Charts

This section contains the ASCII and ANSI character code charts, and the Key code charts that are available on Windows NT and Windows 95 systems. Other character sets are available on OpenVMS and DIGITAL UNIX systems; for details, see the printed *DIGITAL Fortran Language Reference Manual*.

For details on the Fortran 90 character set, see Character Sets.

ASCII Character Codes

The ASCII character code charts contain the decimal and hexadecimal values of the extended ASCII (American Standards Committee for Information Interchange) character set. The extended character set includes the ASCII character set (Chart 1) and 128 other characters for graphics and line drawing (Chart 2), often called the "IBM® character set".

ASCII Character Codes Chart 1

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

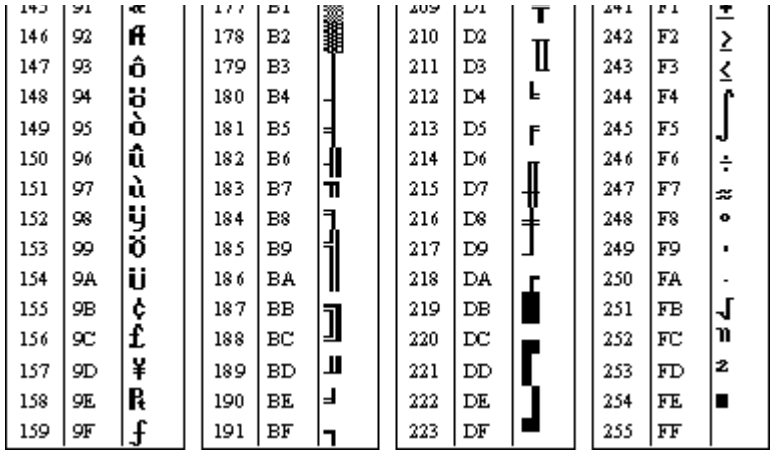
ASCII Character Codes Chart 2 (IBM character set)

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡



ANSI Character Codes

The ANSI character code chart lists the extended character set of most of the programs used by Windows. The codes of the ANSI (American National Standards Institute) character set from 32 through 126 are displayable characters from the ASCII character set. The ANSI characters displayed as solid blocks are undefined characters and may appear differently on output devices.

ANSI Character Codes Chart

0	■	32		64	@	96	`	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	130	‚	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	131	ƒ	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	133	…	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	137	‰	169	©	201	É	233	é
10	■	42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	147	“	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	148	”	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	149	•	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	151	—	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú

27 ■	59 ;	91 [	123 {	‡155 ›	187 »	219 Û	251 û
28 ■	60 <	92 \	124 |	‡156 œ	188 ¼	220 Ü	252 ü
29 ■	61 =	93]	125 }	157 ■	189 ½	221 Ý	253 ý
30 ■	62 >	94 ^	126 ~	158 ■	190 ¾	222 Þ	254 þ
31 ■	63 ?	95 _	127 ■	‡159 Ÿ	191 ¿	223 ß	255 ÿ

■ Indicates that this character isn't supported by Windows.

‡ Indicates that this character is available only in TrueType fonts.

Key Codes

Some keys, such as function keys, cursor keys, and ALT+KEY combinations, have no ASCII code. When a key is pressed, a microprocessor within the keyboard generates an "extended scan code" of two bytes.

The first (low-order) byte contains the ASCII code, if any. The second (high-order) byte has the scan code--a unique code generated by the keyboard when a key is either pressed or released. Because the extended scan code is more extensive than the standard ASCII code, programs can use it to identify keys which do not have an ASCII code.

For more details on key codes, see:

- Key Codes Chart 1
- Key Codes Chart 2

Key Codes Chart 1

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL§
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4$	5	05	52	34	4	36	24	$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E	RS	125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
-_	12	0C	45	2D	-	95	5F	_	31	1F	US	130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F		14	0E	NUL§
TAB	15	0F	9	09		15	0F	NUL	148	94	NUL§	15	A5	NUL§
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	ETB	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
T	20	14	116	74	t	84	54	T	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	TAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B	ESC	26	1A	NUL§
]}	27	1B	93	5D	]	125	7D	}	29	1D	GS	27	1B	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL§
ENTER‡	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL§

LCTRL	29	1D												
RCTRL¢	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOT	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VT	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
;:	39	27	59	3B	;	58	3A	:				39	27	NUL§
'"	40	28	39	27	'	34	22	"				40	28	NUL§
`~	41	29	96	60	`	126	7E	~				41	29	NUL§
LSHIFT	42	2A												
\\|	43	2B	92	5C	\	124	7C	|	28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	ETX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	STX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,<	51	33	44	2C	,	60	3C	<				51	33	NUL§
.>	52	34	46	2E	.	62	3E	>				52	34	NUL§

Key Codes Chart 2

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
/?	53	35	47	2F	/	63	3F	?				53	34	NUL§
GRAY /¢	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
RSHIFT	54	36												
*PRTSC	55	37	42	2A	*	PRTSC		††	16	10				
LALT	56	38												
RALT¢	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11¢	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12¢	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME¢	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP¢	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP¢	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	–						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT¢	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT¢	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END¢	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN¢	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN¢	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS¢	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL

DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL£	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

† Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

§ These key combinations are only recognized on extended keyboards.

£ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.

†† Under MS-DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

Data Representation Models

Several of the numeric intrinsic functions are defined by a model set for integers (for each intrinsic kind used) and reals (for each real kind used). The bit functions are defined by a model set for bits (binary digits).

For more information on the range of values for each data type (and kind), see your programmer's guide.

This section discusses the following topics:

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) 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:

$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$

$i = (-1) \times (4 + 16)$

$i = -1 \times 20$

$i = -20$

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
DIGITAL VAX F_floating [1]	24
IEEE T_floating	53
DIGITAL VAX D_floating [1]	53 [2]
DIGITAL VAX G_floating [1]	53

[1]VMS only
[2]The memory format for VAX D_floating format is 56 mantissa digits, but computationally it is 53 digits. It is considered to have 53 digits by DIGITAL Fortran.

- *e* is an integer in the range e_{min} to e_{max} inclusive. This range differs depending on the real format, as follows:

	e_{min}	e_{max}
IEEE S_floating	-125	128
DIGITAL VAX F_floating [1]	-127	127
IEEE T_floating	-1021	1024
DIGITAL VAX D_floating [1]	-127	127
DIGITAL VAX G_floating [1]	-1023	1023

[1]VMS only

- 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:

$x = 1 \times 2^5 \times (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$

$x = 1 \times 32 \times (.5 + .125)$

$x = 32 \times (.625)$

$x = 20.0$

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:

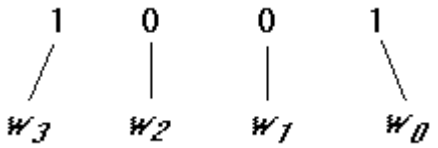
$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

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:



$j = (w_0 \times 2^0) + (w_1 \times 2^1) + (w_2 \times 2^2) + (w_3 \times 2^3)$

$j = 1 + 0 + 0 + 8$

$j = 9$

FORTRAN 77 Syntax

This section contains the syntax for the following features of ANSI FORTRAN 77:

- FORTRAN 77 Data Types
- FORTRAN 77 Intrinsic Functions
- FORTRAN 77 Statements

All are recognized by Visual Fortran without the use of special compiler options, except in certain special instances.

FORTRAN 77 Data Types

The data types defined by ANSI FORTRAN 77 are as follows:

- **INTEGER**
- **REAL**
- **DOUBLE PRECISION**
- **COMPLEX**
- **LOGICAL**
- **CHARACTER** [*n], where *n* is between 1 and 32,767

The data type of a variable, symbolic constant, or function can be declared in a specification statement. If its type is not declared, the compiler determines a data type by the first letter of the variable, constant, or function name. A type statement can also dimension an array variable.

Default requirements for these data types are listed in the following table:

Type	Bytes
INTEGER	4
REAL	4
DOUBLE PRECISION	8
COMPLEX	8
LOGICAL	4
CHARACTER	1
CHARACTER*n*	*n* [1]

[1] Where the maximum *n* is 32,767.

FORTRAN 77 Intrinsic Functions

Function syntax	**Type of return value**
ABS (*gen*)	Same as argument
ACOS (*real*)	Same as argument
AIMAG (*cmp8*)	REAL
AINT (*real*)	Same as argument
ALOG (*real4*)	REAL
ALOG10 (*real4*)	REAL
AMAX0 (*intA, intB* [, *intC*] ..)	REAL
AMAX1 (real4A, real4B, [, real4C]...)	REAL
AMIN0 (*intA, intB* [, *intC*]...)	REAL
AMIN1 (*real4A, real4B* [, *real4C*]...)	REAL
AMOD (*value, mod*)	REAL
ANINT (*value*)	REAL
ASIN (*real*)	Same as argument
ATAN (*real*)	Same as argument
ATAN2 (*realA, realB*)	Same as argument
CABS (*cmp*)	Same as argument; COMPLEX returns REAL
CCOS (*cmp8*)	COMPLEX
CHAR (*int*)	CHARACTER
CLOG (*cmp8*)	COMPLEX
CMPLX (*genA* [, *genB*])	COMPLEX
CONJG (*cx8value*)	COMPLEX
COS (*gen*)	Same as argument
COSH (*real*)	Same as argument
CSIN (*cmp8*)	COMPLEX
CSQRT (*cx8value*)	COMPLEX
DABS (*r8value*)	DOUBLE PRECISION

DACOS (*dbl*)	DOUBLE PRECISION
DASIN (*dbl*)	DOUBLE PRECISION
DATAN (*dbl*)	DOUBLE PRECISION
DATAN2 (*dblA, dblB*)	DOUBLE PRECISION
DBLE (*value*)	DOUBLE PRECISION
DCOS (*dbl*)	DOUBLE PRECISION
DCOSH (*dbl*)	DOUBLE PRECISION
DDIM (dblA, dblB)	DOUBLE PRECISION
DEXP (*dbl*)	DOUBLE PRECISION
DIM (genA, genB)	Same as arguments
DINT (*rvalue*)	DOUBLE PRECISION
DLOG (*dbl*)	DOUBLE PRECISION
DLOG10 (*dbl*)	DOUBLE PRECISION
DMAX1 (*dblA, dblB* [, *dblC*]...)	DOUBLE PRECISION
DMIN1 (*dblA, dblB* [*dblC*]...)	DOUBLE PRECISION
DMOD (*value, mod*)	DOUBLE PRECISION
DNINT (*dbl*)	DOUBLE PRECISION
DPROD (real4A, real4B)	DOUBLE PRECISION
DREAL (*cxvalue*)	DOUBLE PRECISION
DSIGN (dblA, dblB)	DOUBLE PRECISION
DSIN (*dbl*)	DOUBLE PRECISION
DSINH (*dbl*)	DOUBLE PRECISION
DSQRT (*rvalue*)	DOUBLE PRECISION
DTAN (*dbl*)	DOUBLE PRECISION
DTANH (*dbl*)	DOUBLE PRECISION
EXP (*gen*)	Same as argument
FLOAT (*ivalue*)	REAL

IABS (*int*)	Same as argument
ICHAR (*char*)	INTEGER
IDIM (intA, intB)	INTEGER
IDINT (*dbl*)	INTEGER
IDNINT (*dbl*)	INTEGER
IFIX (*real4*)	REAL
INDEX (charA, charB)	INTEGER
INT (*gen*)	INTEGER
ISIGN (*intA, intB*)	INTEGER
LEN (*char*)	INTEGER
LGE (charA, charB)	LOGICAL
LGT (charA, charB)	LOGICAL
LLE (charA, charB)	LOGICAL
LLT (charA, charB)	LOGICAL
LOG (*gen*)	Same as argument
LOG10 (*real*)	Same as argument
MAX (*genA, genB* [, *genC*]...)	INTEGER or REAL
MAX0 (*intA, intB* [, *intC*]...)	INTEGER
MAX1 (*realA, realB* [, *realC*]...)	INTEGER
MIN (*genA, genB* [, *genC*]...)	INTEGER or REAL
MIN0 (*intA, intB* [, *intC*]...)	INTEGER
MIN1 (*realA, real* [, *real*]...)	INTEGER
MOD (genA, genB)	REAL
NINT (*real*)	INTEGER
REAL (*gen*)	REAL
SIGN (genA, genB)	INTEGER or REAL
SIN (*gen*)	Same as argument

SINH (*real*)	Same as argument
SNGL(*dbl*)	REAL
SQRT (*gen*)	Same as argument
TAN (*real*)	Same as argument
TANH (*real*)	Same as argument

FORTRAN 77 Statements

ASSIGN *label* **TO** *variable*

BACKSPACE {*unitspec* |
([**UNIT**=]*unitspec*
[, **ERR**=*errlabel*]
[, **IOSTAT**=*iocheck*])}

BLOCK DATA [*blockdataname*]

CALL *sub* [([*actuals*])]

CHARACTER [*****chars*] *vname* [*****length*] [(*dim*)] [, *vname* [*****length*] [(*dim*)]

CLOSE ([**UNIT=**]*unitspec*
[, **ERR=***errlabel*]
[, **IOSTAT=***iocheck*]
[, **STATUS=***status*])

COMMON [/[*cname*] /] *nlist*,] / [*cname*] /*nlist*] ...

COMPLEX *vnam* [(*dim*)] [,*vname* [(*dim*)]]...

CONTINUE

DATA *nlist* /*clist*/ [[,] *nlist* /*clist*/]...

DIMENSION *array* ([*lower***:**]*upper* [, {[*lower***:**]*upper*])

DO [*label* [,]] *dovar* = *start*, *stop* [, *inc*]

DOUBLE PRECISION *vname* [(*dim*)] [,*vname* [(*dim*)]]...

ELSE
statementblock

ELSE IF (*expression***) THEN**

statementblock

END

END IF

ENDFILE {*unitspec* |
([**UNIT**=] *unitspec*
[, **ERR**=*errlabel*]
[, **IOSTAT**=*iocheck*])}

ENTRY *ename* [([*formal* [,*formal*]...])]

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

EXTERNAL *name* [,*name*]...

FORMAT [*editlist*]

[*type*] **FUNCTION** *func* **(**[*formal*] [,*formal*]...**)**

GOTO *variable* [[**,**] (*labels*)]

GOTO (*labels*) [**,**] *n*

GOTO *label*

IF (*expression*) *label1*, *label2*, *label3*

IF (*expression*) *statement*

IF (*expression*) **THEN**
statementblock1
[**ELSE IF** (*expression*) **THEN**
statementblock2] ...
[**ELSE**
statementblock3]
END IF

IMPLICIT *type* (*letters*) [, *type* (*letters*)]...

INQUIRE ({[**UNIT=**]*unitspec* | **FILE=***file*}
[, **ACCESS=***access*]
[, **BLANK=***blank*] [, **DIRECT=***direct*] [, **ERR=***errlabel*] [, **EXIST=***exist*] [, **FORM=***form*]
[, **FORMATTED=***formatted*] [, **IOSTAT=***iocheck*]
[, **NAME=***name*] [, **NAMED=***named*]
[, **NEXTREC=***nextrec*] [, **NUMBER=***num*] [, **OPENED=***opened*]

[, **RECL=***recl*] [, **SEQUENTIAL=***seq*] [, **UNFORMATTED=***unformatted*])

INTEGER *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

INTRINSIC *names*

LOGICAL *vname* [(*dim*)] [, *vname* [(*dim*)]]...

OPEN ([**UNIT=**]*unitspec* [, **ACCESS=***access*]
[, **BLANK=***blanks*]
[, **ERR=***errlabel*] [, **FILE=***file*]
[, **FORM=***form*] [, **IOSTAT=***iocheck*]
[, **RECL=***recl*] [, **STATUS=***status*])

PARAMETER (*name=constexpr* [, *name=constexpr*]...)

PAUSE [*prompt*]

PRINT { *, | *formatspec* | } [, *iolist*]

PROGRAM *program-name*

READ { *formatspec*, | ([**UNIT=**] *unitspec* [, [**FMT=**]
formatspec] [, **END=***endlabel*] [, **ERR=***errlabel*]
[, **IOSTAT=***iocheck*] [, **REC=***rec*])} *iolist*

REAL *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

RETURN [*ordinal*]

REWIND { *unitspec* |
([**UNIT=**]*unitspec*
[, **ERR=***errlabel*]
[, **IOSTAT=***iocheck*])}

SAVE [*names*]

STOP [*message*]

SUBROUTINE *subr* [([*formal* [, *formal*]...])]

WRITE ([**UNIT=**] *unitspec*
[, [**FMT=**] *formatspec*]
[, **ERR=***errlabel*]
[, **IOSTAT=***iocheck*]
[, **REC=***rec*])
iolist

Summary of Language Extensions

This appendix summarizes the DIGITAL Fortran language extensions to the ANSI/ISO Fortran 90 Standard.

For more information, see the following sections:

- DIGITAL Fortran Language Extensions
- High Performance Fortran Language Extensions

DIGITAL Fortran Language Extensions

This section summarizes the DIGITAL Fortran language extensions. Most extensions are available on all systems, but some extensions are limited to certain systems. If an extension is limited, it is labeled.

Extensions in the following topics are discussed:

- Source Forms
- Names
- Character Sets
- Intrinsic Data Types
- Constants
- Derived Data Types
- Arrays
- Expressions and Assignment
- Specification Statements
- Procedures
- Compilation Control Statements
- Built-In Functions
- I/O Statements
- I/O Formatting
- File Operation Statements
- Compiler Directives
- Additional Language Features
- Intrinsic Procedures

For more information, see High Performance Fortran Language Extensions.

Source Forms

The following are extensions to the methods and rules for source forms:

- Tab-formatting as a method to code lines
- The letter D as a debugging statement indicator in column 1 of fixed or tab source form
- An optional statement field width of 132 columns for fixed or tab source form
- An optional sequence number field for fixed source form

- Up to 511 continuation lines in a source program

Names

The following are extensions to the Fortran 90 rules for names (see names):

- 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 90 character set:

- The Tab (<Tab>) character (see Character Sets)
- The DEC Multinational extension to the ASCII character set (VMS, U*X) [1]
- ASCII Character Code Chart 2--IBM Character Set (WNT, W95)
- ANSI Character Code Chart (WNT, W95)
- Key Code Charts (WNT, W95)

[1] See the printed *DIGITAL Fortran Language Reference Manual*.

Intrinsic Data Types

The following are data-type extensions:

BYTE	INTEGER*1	REAL*8 [2]
LOGICAL*1	INTEGER*2	REAL*16 [3]
LOGICAL*2	INTEGER*4	COMPLEX*8
LOGICAL*4	INTEGER*8 [1]	COMPLEX*16 [2]
LOGICAL*8 [1]	REAL*4	

[1] Alpha only
[2] D_floating and G_floating implementations are available on OpenVMS systems only.
[3] VMS, U*X

For more information, see Intrinsic Data Types.

Constants

Hollerith constants are allowed as an extension.

C Strings are allowed as extensions in character constants.

Derived Data Types

As an extension, default initial values for derived-type components can be specified in a derived-type definition.

Arrays

As an extension, arrays declared using the ALLOCATABLE attribute can be automatically deallocated.

Expressions and Assignment

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary (see Data Type of Numeric Expressions).

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

The following are extensions allowed in logical expressions:

- .XOR. as a synonym for .NEQV.
- Integers as valid logical items

As an extension, the WHERE construct can include nested **WHERE** constructs and a masked **ELSEWHERE** statement. **WHERE** constructs can also be named.

The FORALL construct and statement are extensions.

Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC attribute and statement
- STATIC attribute and statement
- VOLATILE attribute and statement

Procedures

The ELEMENTAL and PURE prefixes are allowed in user-defined functions and subroutines as extensions.

As an extension, the END INTERFACE statement of an interface block defining a generic routine can specify a generic identifier.

Compilation Control Statements

The following statements are extensions that can influence compilation:

- INCLUDE statement format (VMS only):

  INCLUDE '[text-lib] (module-name) [/[NO]LIST]'

- OPTIONS statement:

```
/ASSUME = [NO]UNDERSCORE   (Alpha only)

          {ALL             }
          {[NO]BOUNDS      }
/CHECK = {[NO]OVERFLOW    }
          {[NO]UNDERFLOW   }
          {NONE            }

/NOCHECK

            {BIG_ENDIAN     }
            {CRAY           }
            {FDX            }
            {FGX            }
/CONVERT = {IBM             }
            {LITTLE_ENDIAN  }
            {NATIVE         }
            {VAXD           }
            {VAXG           }

/[NO]EXTEND_SOURCE
/[NO]F77

          {D_FLOAT   (VMS only)  }
/FLOAT = {G_FLOAT   (VMS only)  }
          {IEEE_FLOAT            }

/[NO]G_FLOATING   (VMS only)
/[NO]I4
/[NO]RECURSIVE
```

Built-In Functions

The %VAL, %REF, %DESCR, and %LOC built-in functions are extensions.

I/O Statements

The following I/O statements and specifiers are extensions:

- ACCEPT statement
- REWRITE statement
- **TYPE** statements as synonyms for PRINT statements

- A key-field-value specifier as a control list parameter (VMS only)
- A key-of-reference specifier as a control list parameter (VMS only)
- Indexed **READ** Statement (VMS only)
- Indexed **WRITE** Statement (VMS only)

As an extension, comments (beginning with !) are allowed in namelist input data.

I/O Formatting

The following are extensions allowed in I/O Formatting:

- The Q edit descriptor
- The dollar sign ($) edit descriptor and carriage-control character
- The backslash (\) edit descriptor
- The ASCII NUL carriage-control character
- Variable format expressions
- In output using **I**, **B**, **O**, **Z**, and **F** edit descriptors, the specified value of the field width can be zero. (See General Rules for Numeric Editing.)

File Operation Statements

The following statement specifiers and statements are extensions:

- 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'(VMS only)
  - BINARY (WNT, W95)
  - BUFFERED
  - BLOCKSIZE (WNT, W95)
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN', 'BINARY' (WNT, W95)
  - IOFOCUS (WNT, W95)
  - KEYED (VMS only)
  - MODE as a synonym for ACTION (WNT, W95)
  - ORGANIZATION
  - RECORDTYPE
  - SHARE (WNT, W95)

See also INQUIRE Statement.

- OPEN statement specifiers:
  - ACCESS values: 'KEYED'(VMS only), 'APPEND'
  - ASSOCIATEVARIABLE
  - BLOCKSIZE
  - BUFFERCOUNT
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - DISPOSE
  - EXTENDSIZE (VMS only)
  - FORM value: 'BINARY'(WNT, W95)
  - INITIALSIZE (VMS only)
  - IOFOCUS (WNT, W95)
  - KEY (VMS only)
  - MAXREC
  - MODE as a synonym for ACTION (WNT, W95)
  - NAME as a synonym for FILE
  - NOSPANBLOCKS (VMS only)
  - ORGANIZATION
  - READONLY
  - RECORDSIZE as a synonym for RECL
  - RECORDTYPE
  - SHARE (WNT, W95)
  - SHARED
  - TITLE (WNT, W95)
  - TYPE as a synonym for STATUS
  - USEROPEN

  See also OPEN Statement.

- UNLOCK statement

Compiler Directives

The following General Compiler Directives are extensions:

- **ALIAS**
- **ATTRIBUTES**
- **DECLARE** and **NODECLARE**
- **DEFINE** and **UNDEFINE**
- **FIXEDFORMLINESIZE**
- **FREEFORM** and **NOFREEFORM**
- **IDENT**

- **IF** and **IF DEFINED**
- **INTEGER**
- **MESSAGE**
- **OBJCOMMENT**
- **OPTIONS**
- **PACK**
- **PSECT**
- **REAL**
- **STRICT** and **NOSTRICT**
- **SUBTITLE**
- **TITLE**

Intrinsic Procedures

The following intrinsic procedures are extensions:

ACOSD	CPU_TIME	IDATE	QEXT [4]
ASIND	DATE	ISHA	QFLOAT [4]
ASM [1]	DCMPLX	ISHC	RAN
ATAND	DFLOAT	ISHL	RANDU
ATAN2D	DREAL	ISNAN	SECNDS
CDABS [2]	EOF	LEADZ	SIND
CDCOS [2]	ERRSNS	LOC	SIZEOF
CDEXP [2]	EXIT	MALLOC	TAND
CDLOG [2]	FP_CLASS	MULT_HIGH [2]	TIME
CDSIN [2]	FREE	NULL	TRAILZ
CDSQRT [2]	IARGCOUNT [3]	NWORKERS	ZEXT
COSD	IARGPTR	POPCNT	
COTAN	IBCHNG	POPPAR	

[1] Alpha only

[2] Double precision complex intrinsics can also begin with the letter Z. For example, **CDABS** can also be spelled **ZABS**.

[3] VMS only

[4] VMS, U*X

The following INTEGER(8) specific functions are extensions available on Alpha processors:

AKMAX0	KIBCLR	KIFIX	KMIN0
AKMIN0	KIBITS	KINT	KMIN1
BKTEST	KIBSET	KIOR	KMOD
DFLOTK	KIDIM	KISHFT	KNINT
FLOATK	KIDINT	KISIGN	KNOT
KIABS	KIDNNT	KMAX0	KZEXT
KIAND	KIEOR	KMAX1	

As an extension, the keyword **KIND** can be specified for CEILING and FLOOR.

As an extension, SIGN can distinguish between positive and negative zero.

Additional Language Features

The following are language extensions that facilitate compatibility with other versions of Fortran:

- DEFINE FILE statement
- ENCODE and DECODE statements
- FIND statement
- An alternative syntax for the **PARAMETER** statement
- VIRTUAL statement
- **AND**, **OR**, **XOR**, **IMAG**, **LSHIFT**, **RSHIFT** intrinsics (see the *A to Z Reference*)
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for an I/O record specifier
- An alternate syntax for the **DELETE** statement
- An alternative form for namelist external records
- The DIGITAL Fortran 77 POINTER statement
- Record structures

High Performance Fortran Language Extensions

This section summarizes the High Performance Fortran language extensions to the Fortran 90 standard.

The following extensions are discussed:

- Data Parallel Statements
- Procedure Prefixes
- Intrinsic Procedures

For information on other extensions, see DIGITAL Fortran Language Extensions.

Data Parallel Statements

The following statement and construct are extensions:

- FORALL statement
- **FORALL** construct with multiple assignments

FORALL is also a Fortran 95 feature.

Procedure Prefixes

The following prefixes are allowed in functions and subroutines as extensions:

- PURE - this is also a Fortran 95 feature.
- EXTRINSIC (HPF) - functional only on DIGITAL UNIX systems
- EXTRINSIC (HPF_LOCAL) - functional only on DIGITAL UNIX systems
- EXTRINSIC (HPF_SERIAL) - functional only on DIGITAL UNIX systems

Intrinsic Procedures

System Inquiry Intrinsic Procedures

The following intrinsic procedures are extensions:

- NUMBER_OF_PROCESSORS intrinsic function
- PROCESSORS_SHAPE intrinsic function

Computational Intrinsic Functions

The argument DIM is an extension in the MAXLOC and MINLOC intrinsic functions. This use of argument DIM is also a Fortran 95 feature.

Bit Manipulation Functions

The ILEN intrinsic function is an extension.

A to Z Reference

This section contains the following:

- Language Summary Tables

  This section organizes the functions, subroutines, and statements available in Visual Fortran by the operations they perform. You can use the tables to locate a particular routine for a particular task.

- The descriptions of all Visual Fortran statements and intrinsics, which are listed in alphabetical order.

The Fortran compiler understands statements and intrinsic functions in your program without any additional information, such as that provided in modules.

However, modules must be included in the following types of programs:

- Programs that contain run-time or graphics functions and subroutines must specifically include the Visual Fortran library and graphics modules with the **USE DFLIB** statement.

- Programs that contain Portability procedures must access the Portability library with the **USE DFPORT** statement.

- Programs that use NLS procedures must access the NLS library with the **USE DFNLS** statement.

- Programs that use Dialog procedures must access the Dialog library with the **USE DFLOGM** statement.

- Programs that use Component Object Module (COM) and Automation servers must access the appropriate libraries with the **USE DFCOMTY** statement, as well as the **USE DFCOM** or **USE DFAUTO** statement, whichever is appropriate.

Whenever required, these **USE** module statements are prominent in the *A to Z Reference*.

In addition to the appropriate **USE** statement, you must specify the types of libraries to be used when linking:

- When using the visual development environment, the project type selected and the project settings in the Libraries category (see Categories of Compiler Options) determine the libraries linked against. Also see Errors During the Build Process.

- When using the command line (**DF** command), see Using the Compiler and Linker from the Command Line and Categories of Compiler Options (especially the options under the Libraries category).

Return to the Language Reference Contents

Language Summary Tables

In the following tables, optional arguments for intrinsic procedures are enclosed between brackets: [*optional arg*]. The argument names given in the tables are the keyword names. Keywords allow you to specify optional arguments without regard to order. For example, when invoking **PRODUCT** (ARRAY [, DIM][, MASK]), you can skip the argument DIM by using the statement: array3 = PRODUCT(array1, MASK = array2).

The Fortran procedures and statements have been organized into the following tables:

For more information on keywords, see Argument Keywords in Intrinsic Procedures. For information on using procedures in general, see Program Units and Procedures.

Program Unit Calls and Definitions: table

All the following are statements:

Name	Description
BLOCK DATA	Identifies a block-data subprogram
CALL	Executes a subroutine
COMMON	Delineates variables shared between program units
CONTAINS	Identifies start of a module within a host module
ENTRY	Specifies a secondary entry point to a subroutine or external function
EXTERNAL	Declares a user-defined subroutine or function to be passable as an argument
FUNCTION	Identifies a program unit as a function
INCLUDE	Inserts the contents of a specified file into the source file
INTERFACE	Specifies an explicit interface for external functions and subroutines
INTRINSIC	Declares a predefined function
MODULE	Identifies a module program unit
PROGRAM	Identifies a program unit as a main program
RETURN	Returns control to the program unit that called a subroutine or function
SUBROUTINE	Identifies a program unit as a subroutine
USE	Gives a program unit access to a module

Program Control Statements and Procedures: table

Name	Description
Statements	
CASE	Within a **SELECT CASE** structure, marks a block of statements that are executed if an associated value matches the **SELECT CASE** expression
CONTINUE	Often used as the target of **GOTO** or as the terminal statement in a **DO** loop; performs no operation
CYCLE	Advances control to the end statement of a **DO** loop; the intervening loop statements are not executed
DO	Evaluates statements in the **DO** loop, through and including the ending statement, a specific number of times
DO WHILE	Evaluates statements in the **DO WHILE** loop, through and including the ending

	statement, until a logical condition becomes **.FALSE.**
ELSE	Introduces an **ELSE** block
ELSE IF	Introduces an **ELSE IF** block
ELSEWHERE	Introduces an **ELSEWHERE** block
END	Marks the end of a program unit
END DO	Marks the end of a series of statements following a **DO** or **DO WHILE** statement
END FORALL	Marks the end of a series of statements following a block FORALL statement
END IF	Marks the end of a series of statements following a block IF statement
END SELECT	Marks the end of a SELECT CASE statement
END WHERE	Marks the end of a series of statements following a block WHERE statement
EXIT	Leaves a **DO** loop; execution continues with the first statement following
FORALL	Controls conditional execution of other statements
GOTO	Transfers control to a specified part of the program
IF	Controls conditional execution of other statement(s)
PAUSE	Suspends program execution and, optionally, executes operating-system commands
SELECT CASE	Transfers program control to a block of statements, determined by a controlling argument
STOP	Terminates program execution
WHERE	Controls conditional execution of other statements
Routines	
EXIT	**CALL EXIT**(*exitvalue*). Run-time Subroutine. Terminates the program, flushes and closes all open files, and returns control to the operating system
RAISEQQ	**RAISEQQ**(*sig*). Run-time Function. Sends an interrupt to the executing program, simulating an interrupt from the operating system
SIGNALQQ	**SIGNALQQ**(*sig*, *func*). Run-time Function. Controls signal handling
SLEEPQQ	**CALL SLEEPQQ**(*duration*). Run-time Subroutine. Delays execution of the program for the specified time

Specifying Variables: table

The following are all statements:

Name	Description
AUTOMATIC	Declares a variable on the stack, rather than at a static memory location.
BYTE	Specifies variables as the **BYTE** data type; **BYTE** is equivalent to INTEGER(1).
CHARACTER	Specifies variables as the **CHARACTER** data type.
COMPLEX	Specifies variables as the **COMPLEX** data type.
DATA	Assigns initial values to variables.
DIMENSION	Identifies a variable as an array and specifies the number of elements.
DOUBLE COMPLEX	Specifies variables as the **DOUBLE COMPLEX** data type, equivalent to COMPLEX(8).
DOUBLE PRECISION	Specifies variables as the **DOUBLE-PRECISION** real data type, equivalent to REAL(8).
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory location.
IMPLICIT	Specifies the default typing for real and integer variables and functions.
INTEGER	Specifies variables as the **INTEGER** data type.
LOGICAL	Specifies variables as the **LOGICAL** data type.
MAP...END MAP	Within a **UNION** statement, delimits a group of variable type declarations that are to be ordered contiguously within memory.
NAMELIST	Declares a group name for a set of variables to be read or written in a single statement.
PARAMETER	Equates a constant expression with a name.
REAL	Specifies variables as the **REAL** data type.
RECORD	Declares one or more variables of a user-defined structure type.
SAVE	Causes variables to retain their values between invocations of the procedure in which they are defined.

STATIC	Declares a variable is in a static memory location, rather than on the stack.
STRUCTURE...END STRUCTURE	Defines a new variable type, composed of a collection of other variable types.
TYPE...END TYPE	Defines a new variable type, composed of a collection of other variable types.
UNION...END UNION	Within a structure, causes two or more maps to occupy the same memory locations.
VOLATILE	Specifies that the value of an object is totally unpredictable based on information available to the current program unit.

System, Drive, and Directory Procedures: table

All the following are run-time functions:

Name	Description
CHANGEDIRQQ	**CHANGEDIRQQ**(*dir*). Makes the specified directory the current (default) directory.
CHANGEDRIVEQQ	**CHANGEDRIVEQQ**(*drive*). Makes the specified drive the current drive.
DELDIRQQ	**DELDIRQQ**(*dir*). Deletes a specified directory.
GETDRIVEDIRQQ	**GETDRIVEDIRQQ**(*drivedir*). Returns the current drive and directory path.
GETDRIVESIZEQQ	**GETDRIVESIZEQQ**(*drive*, *total*, *avail*). Gets the size of the specified drive.
GETDRIVESQQ	**GETDRIVESQQ**( ). Reports the drives available to the system.
GETENVQQ	**GETENVQQ**(*varname*, *value*). Gets a value from the current environment.
MAKEDIRQQ	**MAKEDIRQQ**(*dirname*). Makes a directory with the specified directory name.
RUNQQ	**RUNQQ**(*filename*, *commandline)*. Calls another program and waits for it to execute
SETENVQQ	**SETENVQQ**(*varvalue*). Adds a new environment variable, or sets the value of an existing one.
SYSTEMQQ	**SYSTEMQQ**(*commandline*). Executes a command by passing a command string to the operating system’s command interpretor.

File Management: table

Name	Procedure Type	Description
DELFILESQQ	Run-time Function	**DELFILESQQ**(*files*). Deletes the specified files in a specified directory.
FINDFILEQQ	Run-time Function	**FINDFILEQQ**(*filename*, *varname*, *pathbuf*). Searches for a file in the directories specified in the PATH environment variable.
FULLPATHQQ	Run-time Function	**FULLPATHQQ**(*name*, *pathbuf*). Returns the full path for a specified file or directory.
GETDRIVEDIRQQ	Run-time Function	**GETDRIVEDIRQQ**(*drivedir*). Returns current drive and directory path.
GETFILEINFOQQ	Run-time Function	**GETFILEINFOQQ**(*files*, *buffer*, *handle*). Returns information about files with names that match a request string.
PACKTIMEQQ	Run-time Subroutine	**PACKTIMEQQ**(*timedate*, *iyr*, *imon*, *iday*, *ihr*, *imin*, *isec*). Packs time values for use by **SETFILETIMEQQ**.
RENAMEFILEQQ	Run-time Function	**RENAMEFILEQQ**(*oldname*, *newname*). Renames a file.
SETFILEACCESSQQ	Run-time Function	**SETFILEACCESSQQ**(*filename*, *access*). Sets file-access mode for the specified file.
SETFILETIMEQQ	Run-time Function	**SETFILETIMEQQ**(*filename*, *timedate*). Sets modification time for a given file.
SPLITPATHQQ	Run-time Function	**SPLITPATHQQ**(*path*, *drive*, *dir*, *name*, *ext*). Breaks a full path into four components.
UNPACKTIMEQQ	Run-time Subroutine	**UNPACKTIMEQQ**(*timedate*, *iyr*, *imon*, *iday*, *ihr*, *imin*, *isec*). Unpacks a file's packed time and date value into its component parts.

Input/Output Procedures: table

Name	Procedure Type	Description
ACCEPT	Statement	Similar to a formatted, sequential **READ** statement.
BACKSPACE	Statement	Positions a file to the beginning of the previous record.
CLOSE	Statement	Disconnects the specified unit.
DELETE	Statement	Deletes a record from a relative file.
ENDFILE	Statement	Writes an end-of-file record.
EOF	Intrinsic Function	**EOF**(*unit*). Checks for end-of-file record. .TRUE. if at or past end-of-file.
INQUIRE	Statement	Returns the properties of a file or unit.
OPEN	Statement	Associates a unit number with an external device or file.
PRINT (or TYPE)	Statement	Displays data on the screen.
READ	Statement	Transfers data from a file to the items in an I/O list.
REWIND	Statement	Repositions a file to its first record.
REWRITE	Statement	Rewrites the current record.
UNLOCK	Statement	Frees a record in a relative or sequential file that was locked by a previous **READ** statement.
WRITE	Statement	Transfers data from the items in an I/O list to a file

Random Number Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description
RAN	Intrinsic function	result = **RAN**(*i*). Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1.
RANDOM	Run-time Subroutine	**CALL RANDOM**(*ranval*). Returns a pseudorandom real value greater than or equal to zero and less than one.
RANDOM_NUMBER	Intrinsic	**CALL RANDOM_NUMBER**(*harvest*). Returns a

	Subroutine	pseudorandom real value greater than or equal to zero and less than one.
RANDOM_SEED	Intrinsic Subroutine	**CALL RANDOM_SEED**([*size*] [, *put*] [, *get*]). Changes the starting point of **RANDOM_NUMBER**; takes one or no arguments.
RANDU	Intrinsic Subroutine	**CALL RANDU**(*i1*, *i2*, *x*). Computes a pseudorandom number as a single-precision value.
SEED	Run-time Subroutine	**CALL SEED**(*seedval*). Changes the starting point of **RANDOM**.

Date and Time Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description
CPU_TIME	Intrinsic Subroutine	**CALL CPU_TIME** (*time*). Returns the processor time in seconds.
DATE	Intrinsic Subroutine	**CALL DATE** (*buf*). Returns the ASCII representation of the current date (in dd-mmm-yy form).
DATE_AND_TIME	Intrinsic Subroutine	**CALL DATE_AND_TIME**([*date*] [, *time*] [, *zone*] [, *values*]). Returns the date and time.
GETDAT	Run-time Subroutine	**CALL GETDAT** (*iyr*, *imon*, *iday*). Returns the date.
GETTIM	Run-time Subroutine	**CALL GETTIM** (*ihr*, *imin*, *isec*, *i100th*). Returns the time.
IDATE	Intrinsic Subroutine	**CALL IDATE** (*i*, *j*, *k*). Returns three integer values representing the current month, day, and year.
SETDAT	Run-time Function	**SETDAT** (*iyr*, *imon*, *iday*). Sets the date.
SETTIM	Run-time Function	**SETTIM** (*ihr*, *imin*, *isec*, *i100th*). Sets the time.
SYSTEM_CLOCK	Intrinsic Subroutine	**CALL SYSTEM_CLOCK** (*count*, *count_rate*, *count_max*). Returns data from the system clock.
TIME	Intrinsic Subroutine	**CALL TIME** (*buf*). Returns the ASCII representation of the current time (in hh:mm:ss form).

Keyboard and Speaker Procedures: table

Name	Procedure Type	Description
BEEPQQ	Run-time Subroutine	**CALL BEEPQQ**(*freq, duration*). Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.
GETCHARQQ	Run-time Function	**GETCHARQQ**( ). Returns the next keyboard keystroke.
GETSTRQQ	Run-time Function	**GETSTRQQ**(*buffer*). Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Run-time Function	**PEEKCHARQQ**( ). Checks the buffer to see if a keystroke is waiting.

Error Handling: table

Name	Procedure Type	Description
GETLASTERRORQQ	Run-time Function	**GETLASTERRORQQ**( ). Returns the last error set by a run-time function or subroutine.
MATHERRQQ[1]	Run-time Subroutine	**CALL MATHERRQQ**(*name, len, info, retcode*). Replaces default error handling for errors from intrinsic math functions.
SETERRORMODEQQ	Run-time Subroutine	**CALL SETERRORMODEQQ**(*prompt*). Sets the mode for handling critical errors.

[1] x86 only

Argument Inquiry: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
ALLOCATED	Intrinsic Function	**ALLOCATED**(*array*). Determines whether an allocatable array is allocated	*array:* allocatable array result*:* Logical scalar

ASSOCIATED	Intrinsic Function	**ASSOCIATED**(*pointer*[, *target*]). Determines whether a *pointer* and (optional) *target* are associated.	*pointer*: any type *target*: any type result: Logical
DIGITS	Intrinsic Function	**DIGITS**(*x*). Returns number of significant digits for data of the same type as *x*.	*x*: Integer or Real result: Integer
EPSILON	Intrinsic Function	**EPSILON**(*x*). Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as *x*.	*x*: Real result: same type as *x*
GETARG	Run-time Subroutine	**CALL GETARG**(*n, buffer*[, *status*]). Returns the specified command line argument (where the command itself is argument number zero).	*n*: INTEGER(2) or INTEGER(4) *buffer*: Character*(*) *status*: INTEGER(2)
HUGE	Intrinsic Function	**HUGE**(*x*). Returns the largest number that can be represented by numbers of type *x*.	*x*: Integer or Real result: same type as *x*
ILEN	Intrinsic Function	**ILEN**(*i*). Returns the length (in bits) of the two's complement representation of an integer.	*i*: Integer. result: same type as *i*
KIND	Intrinsic Function	**KIND**(*x*). Returns the value of the kind parameter of *x*.	*x*: any intrinsic type result: Integer
LOC	Intrinsic Function	**LOC**(*a*). Returns the address of *a*. *a* can be a variable, function call, expression, or constant.	*a*: any type result: INTEGER(4)
%LOC	Intrinsic Function	Same as **LOC**.	
MAXEXPONENT	Intrinsic Function	**MAXEXPONENT**(*x*). Returns the largest positive decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
MINEXPONENT	Intrinsic Function	**MINEXPONENT**(*x*). Returns the largest negative decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
NARGS	Run-time	**NARGS**( ). Returns the total number	result: INTEGER(4)

	Function	of command-line arguments, including the command.	
PRECISION	Intrinsic Function	**PRECISION**(*x*). Returns the number of significant digits for data of the same type as *x*.	*x*: Real or Complex result: INTEGER(4)
PRESENT	Intrinsic Function	**PRESENT**(*a*). Determines whether an optional argument is present.	*a*: any type result: Logical
RADIX	Intrinsic Function	**RADIX**(*x*). Returns the base for data of the same type as *x*.	*x*: Integer or Real result: INTEGER(4)
RANGE	Intrinsic Function	**RANGE**(*x*). Returns the decimal exponent range for data of the same type as *x*.	*x*: Integer, Real or Complex result: INTEGER(4)
SELECTED_ INT_KIND	Intrinsic Function	**SELECTED_INT_KIND**(*r*). Returns the value of the kind parameter of integers in range *r*.	*r*: Integer result: Integer
SELECTED_ REAL_KIND	Intrinsic Function	**SELECTED_REAL_KIND**([*p*], [*r*]). Returns the value of the kind parameter of reals with (optional) *p* digits and (optional) *r* exponent range. At least one optional argument is required.	*p*: Integer *r*: Integer result: Integer
SIZEOF	Intrinsic Function	**SIZEOF**(*x*). Returns the number of bytes of storage used by the argument.	*x*: any type result: INTEGER(4)
TINY	Intrinsic Function	**TINY**(*x*). Returns the smallest postive number that can be represented by numbers of type *x*.	*x*: Real result: same type as *x*

Memory Allocation and Deallocation Procedures: table

Name	Procedure Type	Description	Argument/Function Type
ALLOCATE	Statement	Dynamically establishes allocatable array dimensions.	
ALLOCATED	Intrinsic	**ALLOCATED**(*array*). Determines	*array:* allocatable

	Function	whether an allocatable array is allocated.	array result*:* Logical scalar
DEALLOCATE	Statement	Frees the storage space previously reserved in an **ALLOCATE** statement.	
FREE	Intrinsic Subroutine	**FREE**(*addr*). Frees the memory block specified by the integer pointer *addr*.	*addr*: INTEGER(4)
MALLOC	Intrinsic Function	**MALLOC**(*size*). Allocates a memory block of size *size* bytes and returns an integer pointer to the block.	*size*: INTEGER(4) result: INTEGER(4)

Array Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
ALL	Intrinsic Function	**ALL**(*mask*[, *dim*]). Determines whether all array values meet the conditions in *mask* along (optional) dimension *dim*.	*mask:* Logical *dim:* Integer result*:* Logical and a scalar if *dim* is absent or *mask* is one-dimensional; otherwise, one dimension smaller than *mask*
ANY	Intrinsic Function	**ANY**(*mask*[, *dim*]). Determines whether any array values meet the conditions in *mask* along (optional) dimension *dim*.	*mask:* Logical *dim:* Integer result*:* Logical and a scalar if *dim* is absent or *mask* is one-dimensional; otherwise, one dimension smaller than *mask*
BSEARCHQQ	Run-time Function	**BSEARCHQQ**(*adr1*, *adr2*, *length*, *size*). Performs a binary search for a specified element on a sorted one-dimensional array of non-structure data types (derived types are not allowed).	*adr1*: INTEGER(4) *adr2*: INTEGER(4) *length*: INTEGER(4) *size*: INTEGER(4) result: INTEGER(4)

COUNT	Intrinsic Function	**COUNT**(*mask*[, *dim*]). Counts the number of array elements that meet the conditions in *mask* along (optional) dimension *dim.*	*mask:* Logical *dim:* Integer result*:* Integer and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, one-dimension smaller than *mask*
CSHIFT	Intrinsic Function	**CSHIFT**(*array*, *shift* [, *dim*]). Performs a circular shift along (optional) dimension *dim.*	*array:* any type *shift:* Integer *dim:* Integer result: same type and shape as *array*
DIMENSION	Statement	Identifies a variable as an array and specifies the number of elements.	
DOT_PRODUCT	Intrinsic Function	**DOT_PRODUCT**(*vector_a*, *vector_b*). Performs dot-product multiplication on vectors (one-dimensional arrays).	*vector_a*: any except Character *vector_b*: same type and size as *vector_a* result: a scalar of the same type as *vector_a*
EOSHIFT	Intinsic Function	**EOSHIFT**(*array*, *shift* [, *boundary*] [, *dim*]). Shifts elements off one end of *array* along (optional) dimension *dim* and copies (optional) *boundary* values in other end.	*array:* any type *shift:* Integer *boundary:* same as *array* *dim:* Integer result: same type and shape as *array*
LBOUND	Intrinsic	**LBOUND**(*array* [, *dim*]).	*array:* any type

	Function	Returns lower dimensional bound(s) of an array along dimension *dim* (optional).	*dim:* Integer result*:* Integer and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, a vector
MATMUL	Intrinsic Function	**MATMUL**(*matrix_a, matrix_b*). Performs matrix multiplication on matrices (two-dimensional arrays).	*matrix_a*: any except Character *matrix_b*: same as *matrix_a* result: same type as *matrix_a*
MAXLOC	Intrinsic Function	**MAXLOC**(*array*[, *dim*] [, *mask*]). Returns the location of the maximum value in an array meeting conditions in (optional) *mask* along optional dimension *dim.*	*array*: Integer or Real *dim:* Integer *mask:* Logical result*:* Integer vector whose size is equal to the number of dimensions in *array*
MAXVAL	Intrinsic Function	**MAXVAL**(*array* [, *dim*] [, *mask*]). Returns the maximum value in an array along (optional) dimension *dim* that meets conditions in (optional) *mask.*	*array:* Integer or Real *dim:* Integer *mask:* Logical result: same type as *array* and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, one dimension smaller than *array*
MERGE	Intrinsic Function	**MERGE**(*tsource*, *fsource*, *mask*). Merges two arrays according to conditions in *mask.*	*tsource:* any type *fsource:* same type and shape as *tsource* *mask:* Logical result: same type and shape as *tsource*
MINLOC	Intrinsic	**MINLOC**(*array* [, *dim*] [,	*array:* Integer or Real

	Function	*mask*]). Returns the location of the minimum value in an array meeting conditions in (optional) *mask* along optional dimension *dim.*	*dim:* Integer *mask:* Logical result*:* Integer vector whose size is equal to the number of dimensions in *array*
MINVAL	Intrinsic Function	**MINVAL**(*array* [, *dim*] [, *mask*]). Returns the minimum value in an array along (optional) dimension *dim* that meets conditions in (optional) *mask.*	*array:* Integer or Real *dim:* Integer *mask:* Logical result: same type as *array* and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, one dimension smaller than *array*
PACK	Intrinsic Function	**PACK**(*array*, *mask* [,*vector*]). Packs an array into a vector (one-dimensional array) of (optional) size *vector* using *mask.*	*array:* any type *mask:* Logical *vector:* same as *array* result: a vector (one-dimensional array) of the same type as *array*
PRODUCT	Intrinsic Function	**PRODUCT**(*array* [, *dim*] [, *mask*]). Returns product of elements of an array along (optional) dimension *dim* that meet conditions in (optional) *mask.*	*array:* Integer, Real or Complex *dim:*Integer *mask:* Logical result: same type as *array* and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, one dimension smaller than *array*
RESHAPE	Intrinsic	**RESHAPE**(*source*, *shape* [,	*source:* any type

	Function	*pad*] [, *order*]). Reshapes an array with subscript *order* (optional), padded with array elements *pad* (optional).	*shape:* Integer *pad:* same as *source* *order:* Integer result: same type as *source* and same shape as *shape*
SHAPE	Intrinsic Function	**SHAPE**(*source*). Returns the shape of an array.	*source:* any type result: a vector (one-dimensional array) of the same type as *source*
SIZE	Intrinsic Function	**SIZE**(*array* [, *dim*]). Returns the extent of *array* along dimension *dim* (optional).	*array:* any type *dim:* Integer result: Integer scalar
SORTQQ	Run-time Subroutine	**CALL SORTQQ**(*addr, count, size*). Sorts a one-dimensional array of non-structure data types (derived types are not allowed).	*addr*: INTEGER(4) *count*: INTEGER(4) *size*: INTEGER(4)
SPREAD	Intrinsic Function	**SPREAD**(*source, dim, ncopies*). Replicates an array by adding a dimension.	*source:* any type *dim:* Integer *ncopies:* Integer result: same type as *source* and one dimension larger
SUM	Intrinsic Function	**SUM**(*array* [, *dim*] [, *mask*]). Sums array elements along dimension *dim* (optional) that meet conditions of *mask* (optional).	*array:* Integer, Real or Complex *dim:* Integer *mask*: Logical result: same type as *array* and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, one dimension smaller than *array*

TRANSPOSE	Intrinsic Function	**TRANSPOSE**(*matrix*). Transposes a two-dimensional array.	*matrix:* any type result: a two-dimensional array the same type as *matrix*
UBOUND	Intrinsic Function	**UBOUND**(*array* [, *dim*]). Returns upper dimensional bound(s) of an array along dimension *dim* (optional).	*array:* any type *dim:* Integer result: same type as *array* and a scalar if *dim* is absent or *array* is one-dimensional; otherwise, a vector.
UNPACK	Intrinsic Function	**UNPACK**(*vector*, *mask*, *field*). Unpacks a vector (one-dimensional array) into an array under *mask* padding with values from *field*.	*vector:* any type *mask:* Logical *field:* same as *vector* result: same type as *vector* and same shape as *mask*

Numeric and Type Conversion Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
ABS	Intrinsic Function	**ABS**(*a*). Returns absolute value of *a*. When **ABS** is passed as an argument, *a* must be REAL(4).	*a*: Integer, Real or Complex result: same type as *a*, except Real for Complex
AIMAG	Intrinsic Function	**AIMAG** (*z*). Returns imaginary part of complex number *z*.	*z*: COMPLEX(4) result: REAL(4)
AINT	Intrinsic Function	**AINT** (*a*[, *kind*]). Truncates *a* to whole number of specified *kind* (optional). When **AINT** is passed as an argument, *a* must be REAL(4).	*a*: Real *kind*: Integer result: Real of type *kind* if present; else same as *a*
AMAX0	Intrinsic	**AMAX0**(*a1*, *a2* [, *a3*...]). Returns	All *a*: INTEGER(4)

	Function	largest value among integer arguments as real.	result: REAL(4)
AMIN0	Intrinsic Function	**AMIN0**(*a1*, *a2* [, *a3*...]). Returns smallest value among integer arguments as real.	All *a*: INTEGER(4) result: REAL(4)
ANINT	Intrinsic Function	**ANINT**(*a* [, *kind*]). Rounds to nearest whole number of specified *kind* (optional). When **ANINT** is passed as an argument, *a* must be REAL(4).	*a*: Real *kind*: Integer result: Real of type *kind* if present; else same as *a*
CEILING	Intrinsic Function	**CEILING**(*a* [, *kind*]). Returns smallest integer greater than *a*.	*a*: Real *kind*: Integer result: INTEGER(4)
CMPLX	Intrinsic Function	**CMPLX**(*x* [,*y*] [,*kind*]). Converts *x* and (optional) *y* to complex of (optional) *kind*.	*x*: Integer, Real, or Complex. *y*: Integer or Real; cannot appear if *x* is complex type *kind*: Integer result: Complex of type *kind* if present; otherwise, single-precision complex
CONJG	Intrinsic Function	**CONJG**(*z*). Returns the conjugate of a complex number.	*z*: COMPLEX(4) result: COMPLEX(4)
DBLE	Intrinsic Function	**DBLE**(*a*). Converts *a* to double precision type.	*a*: Integer, Real, or Complex. result: REAL(8)
DCMPLX	Intrinsic Function	**DCMPLX**(*x* [, *y*]). Converts the argument to double complex type.	*x*: Integer, Real, or Complex. *y*: Integer or Real; cannot appear if x is complex type result: Double complex
DFLOAT	Intrinsic Function	**DFLOAT**(*a*). Converts an integer to double precision type.	*a*: Integer result: REAL(8)
DIM	Intrinsic	**DIM**(*x*, *y*). Returns *x*-*y* if positive;	*x*: Integer or Real

	Function	else 0. When **DIM** is passed as an argument, *a* must be REAL(4).	*y*: same as *x* result: same type as *x*
DPROD	Intrinsic Function	**DPROD**(*x, y*). Returns double-precision product of single precision *x* and *y*.	*x*: REAL(4) *y*: REAL (4) result: REAL(8)
FLOAT	Intrinsic Function	**FLOAT**(*i*). Converts *i* to REAL(4).	*i*: Integer result: REAL(4)
FLOOR	Intrinsic Function	**FLOOR**(*a*[,*kind*]). Returns the greatest integer less than or equal to *a*.	*a*: Real *kind*: Integer result: Integer
IFIX	Intrinsic Function	**IFIX**(*a*). Converts a single-precision real argument to an integer argument by truncating.	*x*: REAL(4) result: Default integer (usually INTEGER(4))
IMAG	Intrinsic Function	Same as AIMAG.	
INT	Intrinsic Function	**INT**(*a* [, *kind*]). Converts a value to integer type.	*a*: Integer, Real, or Complex *kind*: Integer result: Integer of type *kind* if present; else same as *a*
LOGICAL	Intrinsic Function	**LOGICAL**(*l* [, *kind*]). Converts between logical arguments of (optional) *kind*.	*l*: Logical *kind*: Integer
MAX	Intrinsic Function	**MAX**(*a1, a2* [, *a3...*]). Returns largest value among arguments.	All *a*: any type result: same type as *a*
MAX1	Intrinsic Function	**MAX1**(*a1, a2* [, *a3...*]). Returns largest value among real arguments as integer.	All *a*: Real(4) result: Integer
MIN	Intrinsic Function	**MIN**(*a1, a2* [, *a3...*]). Returns largest value among arguments.	All *a*: any type result: same type as *a*
MIN1	Intrinsic	**MIN1**(*a1, a2* [, *a3...*]). Returns	All *a*: REAL(4)

	Function	smallest value among real arguments as integer.	result: Integer
MOD	Intrinsic Function	**MOD**(*a*, *p*). Returns remainder of *a*/*p*. When **MOD** is passed as an argument, *a* must be integer .	*a*: Integer or Real *p*: same as *a* result: same type as *a*
MODULO	Intrinsic Function	**MODULO**(*a*, *p*). Returns *a* modulo *p*.	*a*: Integer or Real *p*: same as *a* result: same type as *a*
NINT	Intrinsic Function	**NINT**(*a* [, *kind*]). Returns the nearest integer to *a*.	*a*: Real *kind*: Integer result: Integer of type *kind* if present; else see Reference entry
REAL	Intrinsic Function	**REAL**(*a* [, *kind*]). Converts a value to real type.	*a*: Integer, Real, or Complex result: REAL(4)
SIGN	Intrinsic Function	**SIGN**(*a*, *b*). Returns absolute value of *a* times the sign of *b*. When **SIGN** is passed as an argument, *a* must be REAL(4).	*a*: Integer or Real *b*: same as *a* result: same type as *a*
SNGL	Intrinsic Function	**SNGL**(*a*). Converts a double-precision argument to single-precision real type.	*a*: REAL(8) result: REAL(4)
TRANSFER	Intrinsic Function	**TRANSFER**(*source*, *mold*[, *size*]). Transforms first argument into type of second argument with (optional) *size* if array.	*source*: any type *mold*: any type *size*: Integer result: same type as *mold*
ZEXT	Intrinsic Function	**ZEXT**(*x*). Extends *x* with zeros.	*a*: Logical or Integer result: Default integer (usually INTEGER(4))

Trigonometric, Exponential, Root, and Logarithmic Procedures: table

Name	Description	Argument/Function Type
ACOS	**ACOS**(*x*). Returns the arc cosine of *x* in radians between 0 and pi. When **ACOS** is passed as an argument, *x* must be REAL(4).	*x:* Real result: same type as *x*
ACOSD	**ACOSD**(*x*). Returns the arc cosine of *x* in degrees between 0 and 180. When **ACOSD** is passed as an argument, *x* must be REAL(4).	*x:* Real result: same type as *x*
ALOG	**ALOG**(*x*). Returns natural log of *x*.	*x*: REAL(4) result: REAL(4)
ALOG10	**ALOG10**(*x*). Returns common log (base 10) of *x*.	*x*: REAL(4) result: REAL(4)
ASIN	**ASIN**(*x*). Returns arc sine of *x* in radians between ±pi/2. When **ASIN** is passed as an argument, *x* must be REAL (4).	*x*: Real result: same type as *x*
ASIND	**ASIND**(*x*). Returns arc sine of *x* in degrees between ±90°. When **ASIND** is passed as an argument, *x* must be REAL (4).	*x*: Real result: same type as *x*
ATAN	**ATAN**(*x*). Returns arc tangent of *x* in radians between ±pi/2. When **ATAN** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
ATAND	**ATAND**(*x*). Returns arc tangent of *x* in degrees between ±90°. When **ATAND** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
ATAN2	**ATAN2**(*y*, *x*). Returns the arc tangent of *y*/*x* in radians between ±pi . When **ATAN2** is passed as an argument, *y* and *x* must be REAL(4).	*y*: Real *x*: same as *y* result: same type as *y*
ATAN2D	**ATAN2D**(*y*, *x*). Returns the arc tangent of *y*/*x* in degrees between ±180°. When **ATAN2D** is passed as an argument, *y* and *x* must be REAL(4).	*y*: Real *x*: same as *y* result: same type as *y*

CCOS	**CCOS**(*x*). Returns complex cosine of *x*.	*x*: COMPLEX(4) result: COMPLEX(4)
CDCOS	**CDCOS**(*x*). Returns double-precision complex cosine of *x*.	*x*: COMPLEX(8) result: COMPLEX(8)
CDEXP	**CDEXP**(*x*). Returns double-precision complex value of *e****x*.	*x*: COMPLEX(8) result: COMPLEX(8)
CDLOG	**CDLOG**(*x*). Returns double-precision complex natural log of *x*.	*x*: COMPLEX(8) result: COMPLEX(8)
CDSIN	**CDSIN**(*x*). Returns double-precision complex sine of *x*.	*x*: COMPLEX(8) result: COMPLEX(8)
CDSQRT	**CDSQRT**(*x*). Returns double-precision complex square root of *x*.	*x* COMPLEX(8) result: COMPLEX(8)
CEXP	**CEXP**(*x*). Returns complex value of *e****x*.	*x*: COMPLEX(4) result: COMPLEX(4)
CLOG	**CLOG**(*x*). Returns complex natural log of *x*.	*x*: COMPLEX(4) result: COMPLEX(4)
COS	**COS**(*x*). Returns cosine of *x* radians. When **COS** is passed as an argument, *x* must be REAL(4).	*x*: Real or Complex result: same type as *x*
COSD	**COSD**(*x*). Returns cosine of *x* degrees. When **COSD** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
COSH	**COSH**(*x*). Returns the hyperbolic cosine of *x*. When **COSH** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
COTAN	**COTAN** (*x*). Returns cotangent of *x* in radians.	*x*: Real result: same type as *x*
COTAND	**COTAND** (*x*). Returns cotangent of *x* in degrees.	*x*: Real result: same type as *x*

CSIN	**CSIN**(*x*). Returns complex sine of *x*.	*x*: COMPLEX(4) result: COMPLEX(4)
CSQRT	**CSQRT**(*x*). Returns complex square root of *x*.	*x*: COMPLEX(4) result: COMPLEX(4)
DACOS	**DACOS**(*x*). Returns double-precision arc cosine of *x* in radians between 0 and pi.	*x*: REAL(8) result: REAL(8)
DACOSD	**DACOSD**(*x*). Returns the arc cosine of *x* in degrees between 0 and 180. When **DACOSD** is passed as an argument, *x* must be REAL(4).	*x*: REAL(8) result: REAL(8)
DASIN	**DASIN**(*x*). Returns double-precision arc sine of *x* in radians between ±pi/2.	*x*: REAL(8) result: REAL(8)
DASIND	**DASIND**(*x*). Returns double-precision arc sine of *x* in degrees between ±90°.	*x*: REAL(8) result: REAL(8)
DATAN	**DATAN**(*x*). Returns double-precision arc tangent of *x* in radians between ±pi/2.	*x*: REAL(8) result: REAL(8)
DATAND	**DATAND**(*x*). Returns double-precision arc tangent of *x* in degrees between ±90°.	*x*: REAL(8) result: REAL(8)
DATAN2	**DATAN2**(*y*, *x*). Returns double-precision arc tangent of *y*/*x* in radians between ±pi.	*y*: REAL(8) *x*: REAL(8) result: REAL(8)
DATAN2D	**DATAN2D**(*y*, *x*). Returns double-precision arc tangent of *y*/*x* in degrees between ±180°.	*y*: REAL(8) *x*: REAL(8) result: REAL(8)
DCOS	**DCOS**(*x*). Returns double-precision cosine of *x* in radians.	*x*: REAL(8) result: REAL(8)
DCOSD	**DCOSD**(*x*). Returns double-precision cosine of *x* in	*x*: REAL(8)

	degrees.	result: REAL(8)
DCOSH	**DCOSH**(*x*). Returns double-precision hyperbolic cosine of *x*.	*x*: REAL(8) result: REAL(8)
DCOTAN	**DCOTAN**(*x*). Returns double-precision cotangent of *x*.	*x*: REAL(8) result: REAL(8)
DEXP	**DEXP**(*x*). Returns double-precision value of *e***x*	*x*: REAL(8) result: REAL(8)
DLOG	**DLOG**(*x*). Returns double-precision natural log of *x*.	*x*: REAL(8) result: REAL(8)
DLOG10	**DLOG10**(*x*). Returns double-precision common log (base 10) of *x*.	*x*: REAL(8) result: REAL(8)
DSIN	**DSIN**(*x*). Returns double-precision sin of *x* in radians.	*x*: REAL(8) result: REAL(8)
DSIND	**DSIND**(*x*). Returns double-precision sin of *x* in degrees.	*x*: REAL(8) result: REAL(8)
DSINH	**DSINH**(*x*). Returns double-precision hyperbolic sine of *x*.	*x*: REAL(8) result: REAL(8)
DSQRT	**DSQRT**(*x*). Returns double-precision square root of *x*.	*x*: REAL(8) result: REAL(8)
DTAN	**DTAN**(*x*). Returns double-precision tangent of *x* in radians.	*x*: REAL(8) result: REAL(8)
DTAND	**DTAND**(*x*). Returns double-precision tangent of *x* in degrees.	*x*: REAL(8) result: REAL(8)
DTANH	**DTANH**(*x*). Returns double-precision hyperbolic tangent of *x*.	*x*: REAL(8) result: REAL(8)
EXP	**EXP**(*x*). Returns value of *e***x*. When **EXP** is passed as an	*x*: Real or Complex

	argument, *x* must be REAL(4).	result: same type as *x*
LOG	**LOG**(*x*) Returns the natural log of *x*.	*x*: Real or Complex result: same type as *x*
LOG10	**LOG10**(*x*). Returns the common log (base 10) of *x*.	*x*: Real result: same type as *x*
SIN	**SIN**(*x*). Returns the sine of *x* radians. When **SIN** is passed as an argument, *x* must be REAL(4).	*x*: Real or Complex result: same type as *x*
SIND	**SIND**(*x*). Returns the sine of *x* degrees. When **SIND** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
SINH	**SINH**(*x*). Returns the hyperbolic sine of *x*. When **SINH** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
SQRT	**SQRT**(*x*). Returns the square root of *x*. When **SQRT** is passed as an argument, *x* must be REAL(4).	*x*: Real or Complex result: same type as *x*
TAN	**TAN**(*x*). Returns the tangent of *x* radians. When **TAN** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
TAND	**TAND**(*x*). Returns the tangent of *x* degrees. When **TAND** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*
TANH	**TANH**(*x*). Returns the hyperbolic tangent of *x*. When **TANH** is passed as an argument, *x* must be REAL(4).	*x*: Real result: same type as *x*

Floating-Point Inquiry and Control Procedures: table

Note: Certain functions (**EXPONENT**, **FRACTION**, **NEAREST**, **RRSPACING**, **SCALE**, **SET_EXPONENT** and **SPACING**) return values related to components of the model set of real numbers. For a description of this model, see the Model for Real Data.

Name	Procedure Type	Description	Argument/Function Type
DIGITS	Intrinsic Function	**DIGITS**(*x*). Returns number of significant digits for data of the same type as *x*.	*x*: Integer or Real result: Integer
EPSILON	Intrinsic Function	**EPSILON**(*x*). Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as *x*.	*x*: Real result: same type as *x*
EXPONENT	Intrinsic Function	**EXPONENT**(*x*). Returns the exponent part of the representation of *x*.	*x:* Real result: Integer
FRACTION	Intrinsic Function	**FRACTION**(*x*). Returns the fractional part of the representation of *x*.	*x*: Real result: same type as *x*
GETCONTROLFPQQ[1]	Run-time Subroutine	**GETCONTROLFPQQ**(*control*). Returns the value of the floating-point processor control word.	*control*: INTEGER (2)
GETSTATUSFPQQ[1]	Run-time Subroutine	**GETSTATUSFPQQ**(*status*). Returns the value of the floating-point processor status word.	*status*: INTEGER(2)
HUGE	Intrinsic Function	**HUGE**(*x*). Returns largest number that can be represented by data of type *x*.	*x*: Integer or Real result: same type as *x*
LCWRQQ[1]	Run-time Subroutine	Same as **SETCONTROLFPQQ**.	
MAXEXPONENT	Intrinsic Function	**MAXEXPONENT**(*x*). Returns the largest positive decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
MINEXPONENT	Intrinsic Function	**MINEXPONENT**(*x*). Returns the largest negative decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
NEAREST	Intrinsic	**NEAREST**(*x*, *s*). Returns the	*x:* Real

	Function	nearest different machine representable number to *x* in the direction of the sign of *s*.	*s*: Real and not zero result: same type as *x*
PRECISION	Intrinsic Function	**PRECISION**(*x*). Returns the number of significant digits for data of the same type as *x*.	*x*: Real or Complex result: INTEGER(4)
RADIX	Intrinsic Function	**RADIX**(*x*). Returns the base for data of the same type as *x*.	*x*: Integer or Real result: INTEGER(4)
RANGE	Intrinsic Function	**RANGE**(*x*). Returns the decimal exponent range for data of the same type as *x*.	*x*: Integer, Real or Complex result: INTEGER(4)
RRSPACING	Intrinsic Function	**RRSPACING**(*x*). Returns the reciprocal of the relative spacing of numbers near *x*.	*x:* Real result: same type as *x*
SCALE	Intrinsic Function	**SCALE**(*x*, *i*). Multiplies *x* by 2 raised to the power of *i*.	*x:* Real *i*: Integer result: same type as *x*
SCWRQQ[1]	Run-time Subroutine	Same as **GETCONTROLFPQQ**.	
SETCONTROLFPQQ[1]	Run-time Subroutine	**SETCONTROLFPQQ** (*controlword*). Sets the value of the floating-point processor control word.	*controlword*: INTEGER(2)
SET_EXPONENT	Intrinsic Function	**SET_EXPONENT**(*x*, *i*). Returns a number whose fractional part is *x* and whose exponential part is *i*.	*x:* Real *i*: Integer result: same type as *x*
SPACING	Intrinsic Function	**SPACING**(*x*). Returns the absolute spacing of numbers near *x*.	*x:* Real result: same type as *x*
SSWRQQ[1]	Run-time Subroutine	Same as **GETSTATUSFPQQ**.	
TINY	Intrinsic	**TINY**(*x*). Returns smallest	*x*: Real

	Function	postive number that can be represented by data of type *x*.	result: same type as *x*

[1] x86 only

See also Miscellaneous Run-Time Procedures: table.

Character Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
ACHAR	Intrinsic Function	**ACHAR**(*i*). Returns character in position *i* in the ASCII sequence	*i*: Integer from 0 to 255 result: CHARACTER(1)
ADJUSTL	Intrinsic Function	**ADJUSTL**(*string*). Adjusts left, removing leading blanks and inserting trailing blanks	*string*: Character*(*) result: same type as *string*
ADJUSTR	Intrinsic Function	**ADJUSTR**(*string*). Adjusts right, removing trailing blanks and inserting leading blanks	*string*: Character*(*) result: same type as *string*
CHAR	Intrinsic Function	**CHAR**(*i* [, *kind*]). Returns character in position *i* in the character sequence of (optional) *kind*	*i*: Integer *kind*: Integer result: CHARACTER(1) of type *kind* if present; otherwise, default kind.
IACHAR	Intrinsic Function	**IACHAR**(*c*). Returns the position of *c* in the ASCII sequence	*c*: CHARACTER(1) result: Integer
ICHAR	Intrinsic Function	**ICHAR**(*c*). Returns the position of *c* in the current character sequence	*c*: CHARACTER(1) result: Integer

INDEX	Intrinsic Function	**INDEX**(*string*, *substring*, *back*]). Returns the starting position of a substring in a string, leftmost or (optional) rightmost occurance	*string*: Character*(*) *substring*: Character*(*) *back*: Logical result: Integer
LEN	Intrinsic Function	**LEN**(*string*). Returns the size of the variable *string*	*string*: Character*(*) result: Integer
LEN_TRIM	Intrinsic Function	**LEN_TRIM**(*string*). Returns the number of characters in *string*, not counting trailing blanks	*string*: Character*(*) result: Integer
LGE	Intrinsic Function	**LGE**(*string_a*, *string_b*). Tests which string comes last in the ASCII sequence; TRUE if equal or *string_a* last	*string_a*: Character*(*) *string_b*: Character*(*) result: Logical
LGT	Intrinsic Function	**LGT**(*string_a*, *string_b*). Tests which string comes last in the ASCII sequence; TRUE if *string_a* last	*string_a*: Character*(*) *string_b*: Character*(*) result: Logical
LLE	Intrinsic Function	**LLE**(*string_a*, *string_b*). Tests which string comes first in the ASCII sequence; TRUE if equal or *string_a* first	*string_a*: Character*(*) *string_b*: Character*(*) result: Logical
LLT	Intrinsic Function	**LLT**(*string_a*, *string_b*). Tests which string comes first in the ASCII sequence; TRUE if *string_a* first	*string_a*: Character*(*) *string_b*: Character*(*) result: Logical
REPEAT	Intrinsic Function	**REPEAT**(*string*, *ncopies*). Concatenates multiple copies of a string	*string*: Character*(*) *ncopies*: Integer result: Character*(*)
SCAN	Intrinsic	**SCAN**(*string*, *set* [, *back*]). Scans a	*string*: Character*(*)

	Function	string for any characters in a set and returns leftmost or (optional) rightmost position where a match is found	*set*: Character*(*) *back*: Logical result: Integer
TRIM	Intrinsic Function	**TRIM**(*string*). Removes trailing blanks from a string	*string*: Character*(*) result: Character*(*)
VERIFY	Intrinsic Function	**VERIFY**(*string*, *set* [, *back*]). Returns the position of the leftmost or (optional) rightmost character in *string* not in *set*, or zero if all characters in*set* are present	*string*: Character*(*) *set*: Character*(*) *back*: Logical result: Integer

Bit Operation and Representation Procedures: table

Note: Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
Bit Operation			
BIT_SIZE	Intrinsic Function	**BIT_SIZE**(*i*). Returns the number of bits in integers of type *i*.	*i*: Integer result: same type as *i*
BTEST	Intrinsic Function	**BTEST**(*i*, *pos*). Tests a bit in position *pos* of *i*; true if bit is 1.	*i*: Integer *pos*: positive Integer result: Logical
IAND	Intrinsic Function	**IAND**(*i*, *j*). Performs a logical AND.	*i*: Integer *j*: Integer result: same type as *i*
IBCHNG	Intrinsic Function	**IBCHNG**(*i*, *pos*). Reverses value of bit in position *pos* of *i*.	*i*: Integer *pos*: positive Integer result: same type as *i*

IBCLR	Intrinsic Function	**IBCLR**(*i, pos*). Clears the bit in position *pos* of *i* to zero.	*i*: Integer *pos*: positive Integer result: same type as *i*
IBITS	Intrinsic Function	**IBITS**(*i, pos, len*). Extracts a sequence of bits of length *len* from *i* starting in position *pos*.	*i*: Integer *pos*: positive Integer *len*: positive Integer result: same type as *i*
IBSET	Intrinsic Function	**IBSET**(*i, pos*). Sets the bit in position *pos* of *i* to one.	*i*: Integer *pos*: positive Integer result: same type as *i*
IEOR	Intrinsic Function	**IEOR**(*i, j*). Performs an exclusive OR.	*i*: Integer *j*: Integer result: same type as *i*
IOR	Intrinsic Function	**IOR**(*i, j*). Performs an inclusive OR.	*i*: Integer *j*: Integer result: same type as *i*
ISHA	Intrinsic Function	**ISHA**(*i, shift*). Shifts *i* arithmetically left or right by *shift* bits; left if *shift* positive, right if *shift* negative. Zeros shifted in from the right, ones shifted in from the left.	*i*: Integer *shift*: Integer result: same type as *i*
ISHC	Intrinsic Function	**ISHC**(*i, shift*). Performs a circular shift of *i* left or right by *shift* bits; left if *shift* positive, right if *shift* negative. No bits lost.	*i*: Integer *shift*: Integer result: same type as *i*
ISHFT	Intrinsic Function	**ISHFT**(*i, shift*). Shifts *i* logically left or right by *shift* bits; left if *shift* positive, right if *shift* negative. Zeros shifted in from opposite end.	*i*: Integer *shift*: Integer result: same type as *i*

ISHFTC	Intrinsic Function	**ISHFTC**(*i*, *shift*[, *size*]). Performs a circular shift of the rightmost bits of (optional) *size* by *shift* bits. No bits lost.	*i*: Integer *shift*: Integer *size*: positive Integer result: same type as *i*
ISHL	Intrinsic Function	**ISHL**(*i*, *shift*). Shifts *i* logically left or right by *shift* bits. Zeros shifted in from opposite end.	*i*: Integer *shift*: Integer result: same type as *i*
MVBITS	Intrinsic Subroutine	**MVBITS**(*from*, *frompos*, *len*, *to*, *topos*). Copies a sequence of bits from one integer to another.	*from*: Integer *frompos*: positive Integer *to*: Integer *topos*: positive Integer
NOT	Intrinsic Function	**NOT**(*i*). Performs a logical complement.	*i*: Integer result: same type as *i*
Bit Representation			
LEADZ	Intrinsic Function	**LEADZ**(*i*). Returns leading zero bits in an integer.	*i*: Integer result: same type as *i*
POPCNT	Intrinsic Function	**POPCNT**(*i*). Returns number of 1 bits in an integer.	*i*: Integer result: same type as *i*
POPPAR	Intrinsic Function	**POPPAR**(*i*). Returns the parity of an integer.	*i*: Integer result: same type as *i*
TRAILZ	Intrinsic Function	**TRAILZ**(*i*). Returns trailing zero bits in an integer.	*i*: Integer result: same type as *i*

QuickWin Procedures: table

Note: Programs that use these procedures must access the appropriate library with USE DFLIB.

Name	Procedure Type	Description
ABOUTBOXQQ	Function	Adds an About Box with customized text.
APPENDMENUQQ	Function	Appends a menu item.
CLICKMENUQQ	Function	Sends menu click messages to the application window.
DELETEMENUQQ	Function	Deletes a menu item.
FOCUSQQ	Function	Makes a child window active, and gives focus to the child window.
GETACTIVEQQ	Function	Gets the unit number of the active child window.
GETHWNDQQ	Function	Gets the true windows handle from window with the specified unit number.
GETWINDOWCONFIG	Function	Returns the current window's properties.
GETWSIZEQQ	Function	Gets the size of the child or frame window.
GETUNITQQ	Function	Gets the unit number corresponding to the specified windows handle. Inverse of **GETHWNDQQ**.
INCHARQQ	Function	Reads a keyboard input and return its ASCII value.
INITIALSETTINGS	Function	Controls initial menu settings and initial frame window settings.
INQFOCUSQQ	Function	Determines which window is active and has the focus.
INSERTMENUQQ	Function	Inserts a menu item.
INTEGERTORGB	Subroutine	Converts a true color value into its red, green and blue components.
MESSAGEBOXQQ	Function	Displays a message box.
MODIFYMENUFLAGSQQ	Function	Modifies a menu item state.
MODIFYMENUROUTINEQQ	Function	Modifies a menu item's callback routine.
MODIFYMENUSTRINGQQ	Function	Changes a menu item's text string.
PASSDIRKEYSQQ	Function	Determines the behavior of direction and page keys.

REGISTERMOUSEEVENT	Function	Registers the application-defined routines to be called on mouse events.
RGBTOINTEGER	Function	Converts a trio of red, green and blue values to a true color value for use with RGB functions and subroutines.
SETACTIVEQQ	Function	Makes the specified window the current active window *without* giving it the focus.
SETMESSAGEQQ	Subroutine	Changes any QuickWin message, including status bar messages, state messages and dialog box messages.
SETWINDOWCONFIG	Function	Configures the current window's properties.
SETWINDOWMENUQQ	Function	Sets the Window menu to which current child window names will be appended.
SETWSIZEQQ	Function	Sets the size of the child or frame window.
UNREGISTERMOUSEEVENT	Function	Removes the callback routine registered by **REGISTERMOUSEEVENT**.
WAITONMOUSEEVENT	Function	Blocks return until a mouse event occurs.

For more information on using these procedures, see Using QuickWin.

Graphics Procedures: table

Note: Programs that use these procedures must access the appropriate library with USE DFLIB.

Name	Procedure Type	Description
ARC, ARC_W	Functions	Draws an arc.
CLEARSCREEN	Subroutine	Clears the screen, viewport, or text window.
DISPLAYCURSOR	Function	Turns the cursor off and on.
ELLIPSE, ELLIPSE_W	Functions	Draws an ellipse or circle.
FLOODFILL, FLOODFILL_W	Functions	Fills an enclosed area of the screen with the current color index, using the current fill mask.
FLOODFILLRGB,	Functions	Fills an enclosed area of the screen with

FLOODFILLRGB_W		the current RGB color, using the current fill mask.
GETARCINFO	Function	Determines the end points of the most recently drawn arc or pie.
GETBKCOLOR	Function	Returns the current background color index.
GETBKCOLORRGB	Function	Returns the current background RGB color.
GETCOLOR	Function	Returns the current color index.
GETCOLORRGB	Function	Returns the current RGB color.
GETCURRENTPOSITION, GETCURRENTPOSITION_W	Subroutines	Returns the coordinates of the current graphics-output position.
GETFILLMASK	Subroutine	Returns the current fill mask.
GETFONTINFO	Function	Returns the current font characteristics.
GETGTEXTEXTENT	Function	Determines the width of the specified text in the current font.
GETGTEXTROTATION	Function	Get the current text rotation angle.
GETIMAGE, GETIMAGE_W	Subroutines	Stores a screen image in memory.
GETLINESTYLE	Function	Returns the current line style.
GETPHYSCOORD	Subroutine	Converts viewport coordinates to physical coordinates.
GETPIXEL, GETPIXEL_W	Functions	Returns a pixel’s color index.
GETPIXELRGB, GETPIXELRGB_W	Functions	Returns a pixel’s RGB color.
GETPIXELS	Function	Returns the color indices of multiple pixels.
GETPIXELSRGB	Function	Returns the RGB colors of multiple pixels.
GETTEXTCOLOR	Function	Returns the current text color index.
GETTEXTCOLORRGB	Function	Returns the current text RGB color.
GETTEXTPOSITION	Subroutine	Returns the current text-output position.
GETTEXTWINDOW	Subroutine	Returns the boundaries of the current text window.

GETVIEWCOORD, GETVIEWCOORD_W	Subroutines	Converts physical or window coordinates to viewport coordinates.
GETWINDOWCOORD	Subroutine	Converts viewport coordinates to window coordinates.
GETWRITEMODE	Function	Returns the logical write mode for lines.
GRSTATUS	Function	Returns the status (success or failure) of the most recently called graphics routine.
IMAGESIZE, IMAGESIZE_W	Functions	Returns image size in bytes.
INITIALIZEFONTS	Function	Initializes the font library.
LINETO, LINETO_W	Functions	Draws a line from the current position to a specified point.
LOADIMAGE, LOADIMAGE_W	Functions	Reads a Windows bitmap file (.BMP) and displays it at the specified location.
MOVETO, MOVETO_W	Subroutines	Moves the current position to the specified point.
OUTGTEXT	Subroutine	Sends text in the current font to the screen at the current position.
OUTTEXT	Subroutine	Sends text to the screen at the current position.
PIE, PIE_W	Functions	Draws a pie slice.
POLYGON, POLYGON_W	Functions	Draws a polygon.
PUTIMAGE, PUTIMAGE_W	Subroutines	Retrieves an image from memory and displays it.
RECTANGLE, RECTANGLE_W	Functions	Draws a rectangle.
REMAPALLPALETTERGB	Function	Remaps a set of RGB color values to indices recognized by the current video configuration.
REMAPPALETTERGB	Function	Remaps a single RGB color value to a color index.
SAVEIMAGE, SAVEIMAGE_W	Functions	Captures a screen image and saves it as a Windows bitmap file.
SCROLLTEXTWINDOW	Subroutine	Scrolls the contents of a text window.
SETBKCOLOR	Function	Sets the current background color.

SETBKCOLORRGB	Function	Sets the current background color to a direct color value rather than an index to a defined palette.
SETCLIPRGN	Subroutine	Limits graphics output to a part of the screen.
SETCOLOR	Function	Sets the current color to a new color index.
SETCOLORRGB	Function	Sets the current color to a direct color value rather than an index to a defined palette.
SETFILLMASK	Subroutine	Changes the current fill mask to a new pattern.
SETFONT	Function	Finds a single font matching the specified characteristics and assigns it to **OUTGTEXT**.
SETGTEXTROTATION	Subroutine	Sets the direction in which text is written to the specified angle.
SETLINESTYLE	Subroutine	Changes the current line style.
SETPIXEL, SETPIXEL_W	Functions	Sets color of a pixel at a specified location.
SETPIXELRGB, SETPIXELRGB_W	Functions	Sets RGB color of a pixel at a specified location.
SETPIXELS	Subroutine	Sets the color indices of multiple pixels.
SETPIXELSRGB	Subroutine	Sets the RGB color of multiple pixels.
SETTEXTCOLOR	Function	Sets the current text color to a new color index.
SETTEXTCOLORRGB	Function	Sets the current text color to a direct color value rather than an index to a defined palette.
SETTEXTPOSITION	Subroutine	Changes the current text position.
SETTEXTWINDOW	Subroutine	Sets the current text display window.
SETVIEWORG	Subroutine	Positions the viewport coordinate origin.
SETVIEWPORT	Subroutine	Defines the size and screen position of the viewport.
SETWINDOW	Function	Defines the window coordinate system.

SETWRITEMODE	Function	Changes the current logical write mode for lines.
WRAPON	Function	Turns line wrapping on or off.

For more information on using these procedures, see Using QuickWin.

Dialog Procedures: table

Note: Programs that use these procedures must access the Dialog library with USE DFLOGM. Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Funct Type
DLGEXIT	Subroutine	**CALL DLGEXIT**(*dlg*). Closes an open dialog.	*dlg*: DIALOG derived type result: Logical
DLGGET	Function	**DLGGET**(*dlg*, *controlid*, *value* [, *index*]). Retrieves values of dialog control variables.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Integer, Logical or Charac *index*: Integer result: Logical
DLGGETCHAR	Function	**DLGGETCHAR**(*dlg*, *controlid*, *value* [, *index*]). Retrieves values of dialog control variables of type Character.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Character *index*: Integer result: Logical
DLGGETINT	Function	**DLGGETINT**(*dlg*, *controlid*, *value*	*dlg*: DIALOG

		[, *index*]). Retrieves values of dialog control variables of type Integer.	derived type *controlid*: Integer *value*: Integer *index*: Integer result: Logical
DLGGETLOG	Function	**DLGGETLOG**(*dlg*, *controlid*, *value* [, *index*]). Retrieves values of dialog control variables of type Logical.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Logical *index*: Integer result: Logical
DLGINIT	Function	**DLGINIT**(*dlgid*, *dlg*). Initializes a dialog.	*dlgid*: Integer *dlg*: DIALOG derived type result: Logical
DLGINITWITH-RESOURCEHANDLE	Function	**DLGINITWITHRESOURCEHANDLE** (*dlgid*, *hinst*, *dlg*). Initializes a dialog.	*dlgid*: Integer *hinst*: Integer *dlg*: DIALOG derived type result: Logical
DLGISDLGMESSAGE	Function	**DLGISDLGMESSAGE**(*mesg*). Determines whether a message is intended for a modeless dialog box and, if it is, processes it.	*mesg*: T_MSG derived type result: Logical
DLGMODAL	Function	**DLGMODAL**(*dlg*). Displays a dialog and processes dialog selections from user.	*dlg*: DIALOG derived type result: Integer
DLGMODELESS	Function	**DLGMODELESS**(*dlg* [, *nCmdShow*,	*dlg*: DIALOG

		hwndParent]). Displays a modeless dialog box.	derived type *nCmdShow*: Integ *hwndParent*: Integ result: Logical
DLGSEND-CTRLMESSAGE	Function	**DLGSENDCTRLMESSAGE**(*dlg*, *controlid*, *msg*, *wparam*, *lparam*). Sends a message to a dialog box control.	*dlg*: DIALOG derived type *controlid*: Integer *msg*: Integer *wparam*: Integer *lparam*: Integer result: Integer
DLGSET	Function	**DLGSET**(*dlg*, *controlid*, *value* [, *index*]). Assigns values to dialog control variables.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Integer, Logical or Charac *index*: Integer result: Logical
DLGSETCHAR	Function	**DLGSETCHAR**(*dlg*, *controlid*, *value* [, *index*]). Assigns values to dialog control variables of type Character.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Character *index*: Integer result: Logical
DLGSETINT	Function	**DLGSETINT**(*dlg*, *controlid*, *value* [,	*dlg*: DIALOG

		index]). Assigns values to dialog control variables of type Integer.	derived type *controlid*: Integer *value*: Integer *index*: Integer result: Logical
DLGSETLOG	Function	**DLGSETLOG**(*dlg*, *controlid*, *value* [, *index*]). Assigns values to dialog control variables of type Logical.	*dlg*: DIALOG derived type *controlid*: Integer *value*: Logical *index*: Integer result: Logical
DLGSETRETURN	Subroutine	**CALL DLGSETRETURN**(*dlg*, *retval*). Sets the return value for **DLGMODAL**.	*dlg*: DIALOG derived type *retval*: Integer
DLGSETSUB	Function	**DLGSETSUB**(*dlg*, *controlid*, *value* [, *index*]). Assigns procedures (callback routines) to dialog controls.	*dlg*: DIALOG derived type *controlid*: Integer *value*: external procedure name *index*: Integer result: Logical
DLGUNINIT	Subroutine	**CALL DLGUNINIT**(*dlg*). Deallocates memory occupied by an initialized dialog.	*dlg*: DIALOG derived type

For more information on using these procedures, see Using Dialogs.

Compiler Directives: table

Note: Each directive name is preceded by the prefix *c***DEC$**; for example, *c***DEC$ ALIAS**. The *c* can be a c, C, !, or *.

Name	Description
ALIAS	Specifies an alternate external name to be used when referring to external subprograms.
ATTRIBUTES	Applies attributes to variables and procedures.
DECLARE	Generates warning messages for undeclared variables.
DEFINE	Creates a variable whose existence can be tested during conditional compilation.
ELSE	Marks the beginning of an alternative conditional-compilation block to an **IF** directive construct.
ELSEIF	Marks the beginning of an alternative conditional-compilation block to an **IF** directive construct.
ENDIF	Marks the end of a conditional-compilation block.
FIXEDFORMLINESIZE	Sets fixed-form line length. This directive has no effect on freeform code.
FREEFORM	Uses freeform format for source code.
IDENT	Specifies an identifier for an object module.
IF	Marks the beginning of a conditional-compilation block.
IF DEFINED	Marks the beginning of a conditional-compilation block.
INTEGER	Selects default integer size.
MESSAGE	Sends a character string to the standard output device.
NODECLARE	(Default) Turns off warning messages for undeclared variables.
NOFREEFORM	(Default) Uses standard FORTRAN 77 code formatting column rules.
NOSTRICT	(Default) Disables a previous **STRICT** directive.
OBJCOMMENT	Specifies a library search path in an object file.
OPTIONS	Controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries.
PACK	Specifies the memory starting addresses of derived-type items.
PSECT	Modifies certain characteristics of a common block.
REAL	Selects default real size.
STRICT	Disables Visual Fortran features not in the Fortran 90 Standard.

SUBTITLE	Prints the specified subtitle on subsequent pages of the source listing.
TITLE	Prints the specified title on subsequent pages of the source listing.
UNDEFINE	Removes a symbolic variable name created with the **DEFINE** directive.

For more information on using these directives, see General Compiler Directives.

National Language Standard Procedures: table

Note: Programs that use these procedures must access the NLS library with USE DFNLS.

Name	Procedure Type	Description
MBCharLen	Function	Returns the length of the first multibyte character in a string.
MBConvertMBToUnicode	Function	Converts a character string from a multibyte codepage to a Unicode string.
MBConvertUnicodeToMB	Function	Converts a Unicode string to a multibyte character string of the current codepage.
MBCurMax	Function	Returns the longest possible mutlibyte character for the current codepage.
MBINCHARQQ	Function	Same as **INCHARQQ**, but can read a single multibyte character at once.
MBINDEX	Function	Same as **INDEX**, except that multibyte characters can be included in its arguments.
MBJISToJMS	Function	Converts a Japan Industry Standard (JIS) character to a Kanji (Shift JIS or JMS) character.
MBJMSToJIS	Function	Converts a Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.
MBLead	Function	Determines whether a given character is the first byte of a multibyte character.
MBLen	Function	Returns the number of multibyte characters in a string, including trailing spaces.
MBLen_Trim	Function	Returns the number of multibyte characters

		in a string, not including trailing spaces.
MBLGE, MBLGT, MBLLE, MBLLT,MBLEQ, MBLNE	Function	Same as **LGE**, **LGT**, **LLE**, and **LLT**, and the logical operators .EQ. and .NE., except that multibyte characters can be included in their arguments.
MBNext	Function	Returns the string position of the first byte of the multibyte character immediately after the given string position.
MBPrev	Function	Returns the string position of the first byte of the multibyte character immediately before the given string position.
MBSCAN	Function	Same as **SCAN**, except that multibyte characters can be included in its arguments.
MBStrLead	Function	Performs a context sensitive test to determine whether a given byte in a character string is a lead byte.
MBVERIFY	Function	Same as **VERIFY**, except that multibyte characters can be included in its arguments.
NLSEnumCodepages	Function	Returns an array of valid codepages for the current console.
NLSEnumLocales	Function	Returns an array of locales (langauge/country combinations) installed on the system.
NLSFormatCurrency	Function	Formats a currency number according to conventions of the current locale (language/country).
NLSFormatDate	Function	Formats a date according to conventions of the current locale (language/country).
NLSFormatNumber	Function	Formats a number according to conventions of the current locale (language/country).
NLSFormatTime	Function	Formats a time according to conventions of the current locale (language/country).
NLSGetEnvironmentCodepage	Function	Returns the current codepage for the system Window or console.
NLSGetLocale	Subroutine	Returns the current language, country, and/or codepage.
NLSGetLocaleInfo	Function	Returns information about the current locale.
NLSSetEnvironmentCodepage	Function	Sets the codepage for the console.

NLSSetLocale	Function	Sets the current language, country, and codepage.

For more information on using these procedures, see Using National Language Support Routines.

See also NLS Date and Time Format.

Portability Procedures: table

Note: Programs that use these procedures must access the portability library with USE DFPORT. Square brackets [...] denote optional arguments.

Name	Procedure Type	Description	Argument/Function Type
ABORT	Subroutine	**CALL ABORT**([*string*]). Flushes and closes all I/O buffers and terminates execution with the optional abort message in *string*.	*string*: CHARACTER*(*)
ACCESS	Function	**ACCESS**(*name*, *mode*). Checks the file specified by *name* for accessibility to the caller in *mode* mode.	*name*: CHARACTER*(*) *mode*: CHARACTER*(*) result: INTEGER(4)
ALARM	Function	**ALARM**(*time*, *proc*). Executes the subroutine *proc* after *time* seconds elapse.	*time*: INTEGER(4) *proc*: External procedure result: INTEGER(4)
BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN	Functions	Bessel functions of the first and second kinds and integer orders. **BESJ0**(*x*), **BESJ1**(*x*), **BESY0**(*x*), **BESY1**(*x*) take argument *x*. **BESJN**(*n*, *x*), **BESYN**(*n*, *x*) take arguments integer order *n* and value *x*.	*n*: INTEGER(4) *x*: REAL(4) result: REAL(4)
BIC, BIS, BIT	Subroutines	Bit clear and set subroutines, and bit	*bitnum*: INTEGER(4)

	and Function	test function. **CALL BIC**(*bitnum*, *target*) clears a bit. **CALL BIS**(*bitnum*, *target*) sets a bit. **BIT**(*bitnum*, *source*) tests a bit.	*target*: INTEGER(4) *source*: INTEGER(4) return from **BIT**: Logical
CHDIR	Function	**CHDIR**(*dir_name*). Changes the default directory to *dir_name*.	*dir_name*: CHARACTER*(*) result: INTEGER(4)
CHMOD	Function	**CHMOD**(*name*, *mode*). Changes the *mode* attributes of a file specified by *name*.	*name*: CHARACTER*(*) *mode*: CHARACTER*(*) result: INTEGER(4)
CLOCK	Function	**CLOCK**( ). Returns the time in HH:MM:SS format.	result: CHARACTER(8)
CTIME	Function	**CTIME**(*stime*). Converts system time to a 24-character ASCII string.	*stime*: INTEGER(4) result: CHARACTER(24)
DATE	Subroutine or Function	**CALL DATE**(*string*) or **DATE**( ). Returns the date as an ASCII string.	*string*: CHARACTER*(*) return from **DATE** function: CHARACTER(8)
DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN	Functions	REAL(8) Bessel functions of the first and second kinds and integer orders. **DBESJ0**(*x*), **DBESJ1**(*x*), **DBESY0**(*x*), **DBESY1**(*x*) take argument *x*. **DBESJN**(*n*, *x*), **DBESYN**(*n*, *x*) take arguments integer order *n* and value *x*.	*n*: INTEGER(4) *x*: REAL(8) result: REAL(8)
DRAND, DRANDM	Functions	**DRAND**(*iflag*) and **DRANDM**(*iflag*). Return random numbers between 0.0 and 1.0, chosen according to the value of *iflag*.	*iflag*: INTEGER(4) result: REAL(8)

DTIME[1]	Function	**DTIME**(*tarray*). Returns the elapsed time since the last call to **DTIME** or the program start.	*tarray(2)*: REAL(4) result: REAL(4)
ETIME[1]	Function	**ETIME**(*tarray*). Returns the elapsed CPU time since the last call to **ETIME** or the program start.	*tarray(2)*: REAL(4) result: REAL(4)
FDATE	Subroutine or Function	**CALL FDATE**(*string*) or **FDATE**( ). Returns the date and time as an ASCII string.	*string*: CHARACTER* (*) return from **FDATE** function: CHARACTER(24)
FGETC	Function	**FGETC**(*lunit*, *char*). Reads the next available character from *lunit* and places it in *char*.	*lunit*: INTEGER(4) *char*: CHARACTER(1) result: INTEGER(4)
FLUSH	Subroutine	**CALL FLUSH**(*lunit*). Causes the contents of the *lunit* buffer to be flushed to the associated file.	*lunit*: INTEGER(4)
FPUTC	Function	**FPUTC**(*lunit*, *char*). Writes a character *char* to the file associated with *lunit*, bypassing normal Fortran I/O.	*lunit*: INTEGER(4) *char*: CHARACTER(1) result: INTEGER(4)
FSEEK	Subroutine	**FSEEK**(*lunit*, *offset*, *from*). Repositions a file on a logical unit to *offset* bytes relative to position *from*.	*lunit*: INTEGER(4) *offset*: INTEGER(4) *from*: INTEGER(4) result: INTEGER(4)
FSTAT	Function	**FSTAT**(*lunit*, *statb*). Returns information about *lunit* in the array *statb*.	*lunit*: INTEGER(4) *statb(12)*: INTEGER(4) result: INTEGER(4)
FTELL	Function	**FTELL**(*lunit*). Returns the current position of the file associated with *lunit* as an offset in bytes from the beginning of the file.	*lunit*: INTEGER(4) result: INTEGER(4)

GERROR	Subroutine	**CALL GERROR**(*string*). Fills *string* with a message for the last detected IERRNO error.	*string*: CHARACTER*(*)
GETC	Function	**GETC**(*char*). Gets the next available character from logical unit 5, usually connected to the console, and places it in *char*, bypassing normal Fortran I/O.	*char*: CHARACTER(1) result: INTEGER(4)
GETCWD	Function	**GETCWD**(*dirname*). Places the current working directory path in *dirname*.	*dirname*: CHARACTER*(*) result: INTEGER(4)
GETENV	Function	**CALL GETENV**(*ename*, *evalue*). Searches the environment list for a string of the form ename=evalue and returns the value found in *evalue* or blanks.	*ename*: CHARACTER*(*) *evalue*: CHARACTER*(*)
GETGID	Function	**GETGID**( ). Included for portability. Always returns 1.	result: INTEGER(4) equal to 1
GETLOG	Subroutine	**CALL GETLOG**(*name*). Returns the user's login name or blanks.	*name*: CHARACTER*(*)
GETPID	Function	**GETPID**( ). Returns the process ID number of the current process.	result: INTEGER(4)
GETUID	Function	**GETUID**( ). Included for portability. Always returns 1.	result: INTEGER(4) equal to 1
GMTIME	Subroutine	**CALL GMTIME**(*stime*, *tarray*). Separates the time returned by **TIME( )** in *stime* into GMT date and time and stores in *tarray*.	*stime*: INTEGER(4) *tarray(9)*: INTEGER(4)
HOSTNAM	Function	**HOSTNAM**(*name*). Puts the name of the current host into *name*.	*name*: CHARACTER*(*) result: INTEGER(4)
IARGC	Function	**IARGC**( ). Returns the index of the last command-line argument.	result: INTEGER(4)
IDATE	Subroutine	**CALL IDATE**(*iarray*) or	*iarray(3)*: INTEGER(4)

		CALL IDATE(*month*, *day*, *year*). Returns the day, month and year in *iarray* or the parameters *month*, *day*, and *year*.	*month*: INTEGER(4) *day*: INTEGER(4) *year*: INTEGER(4)
IERRNO	Function	**IERRNO**( ). Returns the last IERRNO error code.	result: INTEGER(4)
IRAND, IRANDM	Functions	**IRAND**(*iflag*) and **IRANDM**(*iflag*). Return integer random numbers between 0 and (2**31) -1, chosen according to the value of *iflag*.	*iflag*: INTEGER(4) result: INTEGER(4)
ITIME	Subroutine	**CALL ITIME**(*iarray*). Returns the current time in *iarray*.	*iarray(3)*: INTEGER(4)
JDATE	Function	**JDATE**( ). Return the Julian date in the form YYDDD.	result: CHARACTER (8)
KILL	Function	**KILL**(*pid*, *signum*). Sends a signal designated by *signum* (defined in **SIGNAL**) to the calling process designated by *pid* (returned by **GETPID**).	*pid*: INTEGER(4) *signum*: INTEGER(4) result: INTEGER(4)
LNBLNK	Function	**LNBLNK**(*string*). Returns the index of the last nonblank character in *string*.	*string*: CHARACTER* (*) result: INTEGER(4)
LONG	Function	**LONG**(*int2*). Returns an INTEGER (2) argument as an INTEGER(4).	*int2*: INTEGER(2) result: INTEGER(4)
LSTAT	Function	**LSTAT**(*name*, *statb*). Returns information about the file named *name* in the array *statb*.	*name*: CHARACTER* (*) *statb(12)*: INTEGER(4) result: INTEGER(4)
LTIME	Subroutine	**CALL LTIME**(*stime*, *tarray*). Separates the time returned by **TIME( )** in *stime* into the date and time for the local time zone and stores in *tarray*.	*stime*: INTEGER(4) *tarray(9)*: INTEGER(4)
PERROR	Subroutine	**CALL PERROR**(*string*). Sends an	*string*: CHARACTER*

		error message to the standard error stream, preceded by *string* with a message for the last detected IERRNO error.	(*)
PUTC	Function	**PUTC**(*char*).Writes a character *char* to logical unit 6.	*char*: CHARACTER(1) result: INTEGER(4)
QSORT	Subroutine	**CALL QSORT**(*array*, *len*, *isize*, *compar*). Sorts *len* elements of *array* each of length *size* according to the sorting order in a user-supplied function *compar*.	*array*: any type *len*: INTEGER(4) *isize*: INTEGER(4) *compar*: External INTEGER(2) function
RAN	Function	**RAN**(*iseed*). Returns a uniformly distributed random number between 0.0 and 1.0.	*seed*: INTEGER(4) result: REAL(4)
RAND, RANDOM	Functions	**RAND**(*iflag*) and **RANDOM**(*iflag*). Return random numbers between 0.0 and 1.0, chosen according to the value of *iflag*.	*iflag*: INTEGER(4) result: REAL(4)
RENAME	Function	**RENAME**(*from*, *to*). Renames a file from *from* to *to*. If *to* exists it is removed first.	*from*: CHARACTER*(*) *to*: CHARACTER*(*) result: INTEGER(4)
RINDEX	Function	**RINDEX**(*string*, *substr*). Returns the index of the last occurrence of *substr* in *string*, or 0.	*string*: CHARACTER*(*) *substr*: CHARACTER*(*) result: INTEGER(4)
RTC	Function	**RTC**( ). Returns the number of seconds since 00:00:00 Greenwich mean time, January 1, 1970.	result: REAL(8)
SECNDS	Function	**SECNDS**(*offset*). Returns the number of seconds that have elapsed since midnight, minus *offset*.	*offset*: REAL(4) result: REAL(4)
SHORT	Function	**SHORT**(*int4*). Returns an	*int4*: INTEGER(4)

		INTEGER(4) argument as an INTEGER(2).	result: INTEGER(2)
SIGNAL	Function	**SIGNAL**(*signum*, *proc*, *flag*). Changes the action taken for the signal designated by *signum* to the external signal processing procedure *proc*. Implementation of *proc* is controlled by *flag*.	*signum*: INTEGER(4) *proc*: External function *flag*: INTEGER(4) result: INTEGER(4)
SLEEP	Subroutine	**CALL SLEEP**(*itime*). Suspends the calling process for *itime* seconds.	*itime*: INTEGER(4)
STAT	Function	**STAT**(*name*, *statb*). Returns information about the file named *name* in the array *statb*.	*name*: CHARACTER*(*) *statb(12)*: INTEGER(4) result: INTEGER(4)
SYSTEM	Function	**SYSTEM**(*string*). Causes *string* to be given to your shell as input as if *string* had been typed as a command.	*string*: CHARACTER*(*) result: INTEGER(4)
TIME	Subroutine or Function	**CALL TIME**(*string*) or **TIME**( ). As a subroutine, fills *string* with the current time formatted as HH:MM:SS. As a function, returns elapsed seconds since 00:00:00 Greenwich mean time, January 1, 1970.	*string*: CHARACTER(8) result: INTEGER(4)
TIMEF	Function	**TIMEF**( ). Returns the number of seconds since the first time it was called. The first time called, **TIMEF** returns 0.0D0.	result: REAL(8)
UNLINK	Function	**UNLINK**(*name*). Removes the file specified by path *name*.	*name*: CHARACTER*(*) result: INTEGER(4)

[1] WNT only

Warning: The two-digit year return value of **DATE**, **IDATE**, and **JDATE** may cause problems with the year 2000. Use DATE_AND_TIME instead.

For more information on using these procedures, see Portability Library in the *Programmer's Guide*.

COM and Automation Procedures: table

Note: Programs that use COM procedures must access the appropriate libraries with USE DFCOM and USE DFCOMTY. Programs that use Automation procedures must access the appropriate libraries with USE DFAUTO and USE DFCOMTY.

Component Object Model (COM) Procedures (DFCOM)		
Name	**Procedure Type**	**Description**
COMAddObjectReference	Function	Adds a reference to an object's interface.
COMCLSIDFromProgID	Subroutine	Passes a programmatic identifier and returns the corresponding class identifier.
COMCLSIDFromString	Subroutine	Passes a class identifier string and returns the corresponding class identifier.
COMCreateObjectByGUID	Subroutine	Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.
COMCreateObjectByProgID	Subroutine	Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.
COMGetActiveObjectByGUID	Subroutine	Passes a class identifier and returns a pointer to the interface of a currently active object.
COMGetActiveObjectByProgID	Subroutine	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
COMInitialize	Subroutine	Initializes the COM library.
COMGetFileObject	Subroutine	Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
COMQueryInterface	Subroutine	Passes an interface identifier and returns a pointer to an object's interface.
COMReleaseObject	Function	Indicates that the program is done with a reference to an object's interface.
COMUninitialize	Subroutine	Uninitializes the COM library.

Automation Server Procedures (DFAUTO)		
Name	**Procedure Type**	**Description**
AUTOAddArg	Subroutine	Passes an argument name and value and adds the argument to the argument list data structure.
AUTOAllocateInvokeArgs	Function	Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.
AUTODeallocateInvokeArgs	Subroutine	Deallocates an argument list data structure.
AUTOGetExceptInfo	Subroutine	Retrieves the exception information when a method has returned an exception status.
AUTOGetProperty	Function	Passes the name or identifier of the property and gets the value of the Automation object's property.
AUTOGetPropertyByID	Function	Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.
AUTOGetPropertyInvokeArgs	Function	Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.
AUTOInvoke	Function	Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.
AUTOSetProperty	Function	Passes the name or identifier of the property and a value, and sets the value of the Automation object's property.
AUTOSetPropertyByID	Function	Passes the member ID of the property and sets the value of the Automation object's property, using the argument list's first argument.
AUTOSetPropertyInvokeArgs	Function	Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

For more information on using these procedures, see Using COM and Automation Objects.

Miscellaneous Run-Time Procedures: table

Name	Procedure Type	Description
FOR_CHECK_FLAWED_PENTIUM	Function	**FOR_CHECK_FLAWED_PENTIUM** ( ). Checks the processor to determine if it shows characteristics of the Pentium® floating-point divide flaw.
FOR_GET_FPE	Function	**FOR_GET_FPE** ( ). Returns the current settings of floating-point exception flags.
FOR_RTL_FINISH_	Function	**FOR_RTL_FINISH_** ( ). Cleans up the Fortran run-time environment; for example, flushing buffers and closing files. It also issues messages about floating-point exceptions, if any occur.
FOR_RTL_INIT_	Subroutine	**FOR_RTL_INIT_** (*argcount*, *actarg*). Initializes the Fortran run-time environment. It establishes handlers and floating-point exception handling, so Fortran subroutines behave the same as when called from a Fortran main program.
FOR_SET_FPE	Function	**FOR_SET_FPE** (*a*). Sets the floating-point exception flags.
FOR_SET_REENTRANCY	Function	**FOR_SET_REENTRANCY** (*mode*). Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits.

Functions Not Allowed as Actual Arguments: table

The following specific functions cannot be passed as actual arguments:

AIMAX0	EOF	JIDINT	LOC
AIMIN0	FLOAT	JIFIX	MALLOC
AJMAX0	FLOATI	JINT	MAX0
AJMIN0	FLOATJ	JMAX0	MAX1
AKMAX0	FLOATK	JMAX1	MIN0
AKMIN0	ICHAR	JMIN0	MIN1
AMAX0	IDINT	JMIN1	MULT_HIGH

AMAX1	IFIX	KIDINT	NUMBER_OF_PROCESSORS
AMIN0	IIDINT	KIFIX	NWORKERS
AMIN1	IIFIX	KINT	PROCESSORS_SHAPE
CHAR	IINT	KIQINT	QEXT
DBLE	IMAX0	KIQNNT	QEXTD
DBLEQ	IMAX1	KMAX0	QMAX1
DFLOTI	IMIN0	KMAX1	QMIN1
DFLOTJ	IMIN1	KMIN0	RAN
DFLOTK	INT	KMIN1	REAL
DMAX1	INT1	LGE	SECNDS
DMIN1	INT2	LGT	SIZEOF
DPROD	INT4	LLE	SNGL
DREAL	JFIX	LLT	SNGLQ

ABORT

Portability Subroutine: Flushes and closes I/O buffers, and terminates program execution.

Module: USE DFPORT

Syntax

CALL ABORT [*string*]

string
(Input; optional) Character*(*). Allows you to specify an abort message at program termination. When **ABORT** is called, "abort:" is written to external unit 0, followed by *string*. If omitted, the default message written to external unit 0 is "abort: Fortran Abort Called."

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EXIT, STOP

Example

```
!The following prints "abort: Fortran Abort Called"
CALL ABORT

!The following prints "abort: Out of here!"
Call ABORT ("Out of here!")
```

ABOUTBOXQQ

QuickWin Function: Specifies the information displayed in the message box that appears when the user selects the About command from a QuickWin application’s Help menu.

Module: USE DFLIB

Syntax

result = **ABOUTBOXQQ** (*cstring*)

cstring
(Input; output) Character*(*). Null-terminated C string.

Results:

The value of the result is INTEGER(4). It is zero if successful; otherwise, nonzero.

If your program does not call **ABOUTBOXQQ**, the QuickWin run-time library supplies a default

string. For further discussion, see Using QuickWin in the *Programmer's Guide*.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin

Example

```
        USE DFLIB
        INTEGER(4) dummy
! Set the About box message
        dummy = ABOUTBOXQQ ('Matrix Multiplier\r      Version 1.0'C)
```

ABS

Elemental Intrinsic Function (Generic): Computes an absolute value.

Syntax

result = **ABS** (*a*)

a
(Input) 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 [2]	INTEGER(8)	INTEGER(8)
ABS	REAL(4)	REAL(4)
DABS	REAL(8)	REAL(8)
QABS [3]	REAL(16)	REAL(16)
CABS [4]	COMPLEX(4)	REAL(4)
CDABS [5]	COMPLEX(8)	REAL(8)

[1] Or JIABS. For compatibility with older versions of Fortran, IABS can also be specified as a generic function.
[2] VMS and U*X
[3] Alpha only
[4] The setting of compiler option /real_size can affect CABS.
[5] This function can also be specified as ZABS.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

ABS (-7.4) has the value 7.4.

ABS ((6.0, 8.0)) has the value 10.0.

The following ABS.F90 program calculates two square roots, retaining the sign:

```
        REAL mag(2), sgn(2), result(2)
        WRITE (*, '(A)') ' Enter two signed magnitudes: '
        READ (*, *) mag
        sgn = SIGN((/1.0, 1.0/), mag) ! transfer the signs to 1.0s
        result = SQRT (ABS (mag))
! Restore the sign by multiplying by -1 or +1:
        result = result * sgn
        WRITE (*, *) result
        END
```

ACCEPT

Statement: A data transfer input statement. It is the same as a formatted, sequential **READ** statement, except that an **ACCEPT** statement must never be commented to user-specified I/O units.

Syntax

Formatted
ACCEPT *form* [, *io-list*]

Formatted: List-Directed
ACCEPT * [, *io-list*]

Formatted: Namelist
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.

Example

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)
```

ACCESS

Portability Function: Determines if a file exists and how it can be accessed.

Module: USE DFPORT

Syntax

result = **ACCESS** (*name*, *mode*)

name
(Input) Character*(*). Name of the file whose accessibility is to be determined.

mode
(Input) Character*(*). Modes of accessibility to check for. Must be a character string of length one or greater containing only the characters "r", "w", "x", or "" (a blank). These characters are interpreted as follows.

Character	Meaning
r	Tests for read permission
w	Tests for write permission
x	Tests for execute permission (name must be .COM, .EXE, .BAT, or .CMD)
(blank)	Tests for existence

The characters within *mode* can appear in any order or combination. For example, wrx and r are legal forms of *mode* and represent the same set of inquiries.

Results:

The value of the result is INTEGER(4). It is zero if all inquiries specified by *mode* are affirmative. If

either argument is illegal, or if the file cannot be accessed in all of the modes specified, one of the following error codes is returned:

- EACCES: Access denied; the file's permission setting does not allow the specified access
- EINVAL: The mode argument is invalid
- ENOENT: File or path not found

For a list of error codes, see IERRNO.

The *name* argument can contain either forward or backward slashes for path separators.

Note that all files are readable. A test for read permission always returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INQUIRE, GETFILEINFOQQ

Example

```
! checks for read and write permission on the file "DATAFILE.TXT"

J = ACCESS ("DATAFILE.TXT", "rw")
! checks whether "DATAFILE.TXT" is executable. It is not, since
! it does not end in .COM, .EXE, .BAT, or .CMD
J = ACCESS ("DATAFILE.TXT","x")
```

ACHAR

Elemental Intrinsic Function (Generic): Returns the character in a specified position of the ASCII character set. It is the inverse of the IACHAR function.

Syntax

result = **ACHAR** (*i*)

i
(Input) Is of type integer.

Results

The result type is character of length 1 with 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. For a complete list of ASCII character codes, see Character and Key Code Charts.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CHAR, IACHAR, ICHAR

Examples

ACHAR (71) has the value `'G'`.

ACHAR (63) has the value `'?'`.

ACOS

Elemental Intrinsic Function (Generic): Produces an arccosine (with the result in radians).

Syntax

result = **ACOS** (*x*)

x
(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results:

The result type is the same as *x*. The value lies in the range 0 to pi.

Specific Name	Argument Type	Result Type
ACOS	REAL(4)	REAL(4)
DACOS	REAL(8)	REAL(8)
QACOS [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ACOS (0.68032123) has the value .8225955.

ACOSD

Elemental Intrinsic Function (Generic): Produces an arccosine (with the result in degrees).

Syntax

result = **ACOSD** (*x*)

x
(Input) 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*.

Specific Name	Argument Type	Result Type
ACOSD	REAL(4)	REAL(4)
DACOSD	REAL(8)	REAL(8)
QACOSD [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ACOSD (0.886579) has the value 27.55354.

ADJUSTL

Elemental Intrinsic Function (Generic): Adjusts a character string to the left, removing leading blanks and inserting trailing blanks.

Syntax

result = **ADJUSTL** (*string*)

string
(Input) Must be of type character.

Results:

The result type is 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ADJUSTR

Examples

```
CHARACTER(16) STRING
STRING= ADJUSTL('   Fortran 90   ') ! returns 'Fortran 90   '

ADJUSTL ('    SUMMERTIME')     ! has the value 'SUMMERTIME    '
```

ADJUSTR

Elemental Intrinsic Function (Generic): Adjusts a character string to the right, removing trailing blanks and inserting leading blanks.

Syntax

result = **ADJUSTR** (*string*)

string
(Input) Must be of type character.

Results:

The result type is 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ADJUSTL

Example

```
CHARACTER(16) STRING
STRING= ADJUSTR('   Fortran 90   ') ! returns '     Fortran 90'

ADJUSTR ('SUMMERTIME----')     ! has the value '----SUMMERTIME'
```

AIMAG

Elemental Intrinsic Function (Generic): Returns the imaginary part of a complex number. This function can also be specified as **IMAG**.

Syntax

result = **AIMAG** (*z*)

z
(Input) Must be of type complex.

Results:

The result type is real with the same kind parameter as *z*. If *z* has the value (x, y), the result has the value y.

The setting of compiler option /real_size can affect **AIMAG**.

Specific Name	Argument Type	Result Type
AIMAG	COMPLEX(4)	REAL(4)
DIMAG	COMPLEX(8)	REAL(8)

To return the real part of complex numbers, use REAL.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CONJG, DBLE

Examples

AIMAG ((4.0, 5.0)) has the value 5.0.

The program AIMAG.F90 applies the quadratic formula to a polynomial and allows for complex results:

```
REAL a, b, c
COMPLEX ans1, ans2, d
WRITE ( *, 100)
100 FORMAT (’  Enter A, b, and c of the  ’,   &
                          ’polynomial ax**2 + bx + c:  ’\)
READ (*, *) a, b, c
d = CSQRT (CMPLX (b**2 - 4.0*a*c))  ! d is either:
                                    !  0.0 + i root, or
                                    !  root + i 0.0
ans1 = (-b + d) / (2.0 * a)
ans2 = (-b + d) / (2.0 * a)
WRITE (*, 200)
200 FORMAT (/ ’  The roots are:’  /)
WRITE (*, 300) REAL(ans1), AIMAG(ans1), &
                   REAL(ans2), AIMAG(ans2)
300 FORMAT (’ X = ’, F10.5, ’ + i’, F10.5)
END
```

AINT

Elemental Intrinsic Function (Generic): Truncates a value to a whole number.

Syntax

result = **AINT** (*a* [, *kind*])

a
(Input) Must be of type real.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of *a*.

The *a* 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

To round rather than truncate, use ANINT.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

AINT (3.678) has the value 3.0.

AINT (-1.375) has the value -1.0.

```
REAL r1, r2
REAL(8) r3(2)
r1 = AINT(2.6)   ! returns the value 2.0
r2 = AINT(-2.6)  ! returns the value -2.0
r3 = AINT((/1.3, 1.9/), KIND = 8)       ! returns the values
                                        ! (1.0D0, 1.0D0)
```

ALARM

Portability Function: Causes a subroutine to begin execution after a specified amount of time has elapsed.

Module: USE DFPORT

Syntax

result = **ALARM** (*time*, *proc*)

time
(Input) Integer. Specifies the time delay, in seconds, between the call to **ALARM** and the time when *proc* is to begin execution. If *time* is 0, the alarm is turned off and no routine is called.

proc
(Input) Name of the procedure to call, which takes no arguments.

Results:

The return value is INTEGER(4). It is zero if no alarm is pending. If an alarm is pending (has already been set by a previous call to **ALARM**), it returns the number of seconds remaining until the prevoiusly set alarm is to go off, rounded up to the nearest second.

After **ALARM** is called and the timer starts, the calling program continues for *time* seconds. The calling program then suspends and calls *proc*, which runs in another thread. When *proc* finishes, the alarm thread terminates, the original thread resumes, and the calling program resets the alarm. Once the alarm goes off, it is disabled until set again.

If *proc* performs I/O or otherwise uses the Fortran library, you need to compile it with one of the multithread libraries. For more information on multithreading, see Creating Multithread Applications in the *Programmer's Guide*.

The thread that *proc* runs in has a higher priority than any other thread in the process. All other threads are essentially suspended until *proc* terminates, or is blocked on some other event, such as I/O.

No alarms can occur after the main process ends. If the main program finishes or any thread executes an **EXIT** call, than any pending alarm is deactivated before it has a chance to run.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RUNQQ

Example

```
USE DFPORT
INTEGER(4) numsec, istat
```

```
EXTERNAL subprog
numsec = 4
write *, "subprog will begin in ", numsec, " seconds"
ISTAT = ALARM (numsec, subprog)
```

ALIAS

Compiler Directive: Declares alternate external names for external subprograms.

Syntax

*c***DEC$ ALIAS** *internal-name, external-name*

c
Is a c, C, !, or *. (See Syntax Rules for General Directives.)

internal-name
The name of the subprogram as used in the current program unit.

external-name
A name or a character constant, delimited by apostrophes or quotation marks.

Rules and Behavior

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.

You can use the prefix !MS$ in place of cDEC$.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ATTRIBUTES - ALIAS option, General Compiler Directives

ALL

Transformational Intrinsic Function (Generic): Determines if *all* values are true in an entire array or in a specified dimension of an array.

Syntax

result = **ALL** (*mask* [, *dim*])

mask
(Input) Must be a logical array.

dim
(Optional; input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ANY, COUNT

Examples

```
        LOGICAL mask( 2, 3), AR1(3), AR2(2)
        mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., .FALSE., &
                        .FALSE./),(/2,3/))
! mask is true false false
! true true false
        AR1 = ALL(mask,DIM = 1) ! evaluates the elements column by
                ! column yielding [true false false]
        AR2 = ALL(mask,DIM = 2) ! evaluates the elements row by row
                ! yielding [false false].
```

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

```
[ 1  5  7 ]
[ 3  6  8 ]
```

and B is the array

```
[ 0  5  7 ]
[ 2  6  9 ].
```

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.

ALLOCATABLE

Statement and 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 Declaration Statement:

type, [*att-ls*,] **ALLOCATABLE** [*att-ls*,] :: *a*[(*d-spec*)] [, *a*[(*d-spec*)]]...

Statement:

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Type declaration statements, Compatible attributes, DEALLOCATE, Arrays, Allocation of Allocatable Arrays, SAVE

Examples

```
!Method for creating and allocating deferred-shape arrays.
INTEGER, ALLOCATABLE :: matrix(:,:)
REAL, ALLOCATABLE    :: vector(:)
...
ALLOCATE(matrix(3,5),vector(-2:N+2))
...
```

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
```

ALLOCATE

Statement: Dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized.

Syntax

ALLOCATE (*object* [(*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
(Output) 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.

To release the storage for an allocated array, use **DEALLOCATE**.

To determine whether an allocatable array is currently allocated, use the **ALLOCATED** intrinsic function.

To determine whether a pointer is currently associated with a target, use the **ASSOCIATED** intrinsic function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATABLE, ALLOCATED, DEALLOCATE, ASSOCIATED, POINTER, Dynamic Allocation

Examples

```
!Method for creating and allocating deferred shape arrays.
INTEGER,ALLOCATABLE::matrix(:,:)
REAL, ALLOCATABLE:: vector(:)
. . .
ALLOCATE (matrix(3,5),vector(-2:N+2))
. . .
```

The following is another 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)
```

ALLOCATED

Inquiry Intrinsic Function (Generic): Indicates whether an allocatable array is currently allocated.

Syntax

ALLOCATED (*array*)

array
(Input) Must be an allocatable array.

Results:

The result is a default logical scalar.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATABLE, ALLOCATE, DEALLOCATE, Arrays, Dynamic Allocation

Examples

```
REAL, ALLOCATABLE :: A(:)
...
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

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
```

AND

Elemental Intrinsic Function (Generic): See IAND

Example

```
INTEGER(1) i, m
```

```
INTEGER result
INTEGER(2) result2
i = 1
m = 3
result = AND(i,m) ! returns an integer of default type
        ! (INTEGER(4) unless reset by user) whose
        ! value = 1
result2 = AND(i,m) ! returns an INTEGER(2) with value = 1
```

ANINT

Elemental Intrinsic Function (Generic): Calculates the nearest whole number.

Syntax

result = **ANINT** (*a* [, *kind*])

a
(Input) Must be of type real.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

To truncate rather than round, use AINT.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NINT

Examples

```
REAL r1, r2
```

```
       r1 = ANINT(2.6)     ! returns the value 3.0
       r2 = ANINT(-2.6)    ! returns the value -3.0

! ANINT.F90 Calculates and adds tax to a purchase amount.
       REAL amount, taxrate, tax, total
       taxrate = 0.081
       amount = 12.99
       tax = ANINT (amount * taxrate * 100.0) / 100.0
       total = amount + tax
       WRITE (*, 100) amount, tax, total
  100  FORMAT ( 1X, 'AMOUNT', F7.2 /
      + 1X, 'TAX ', F7.2 /
      + 1X, 'TOTAL ', F7.2)
       END
```

ANINT (3.456) has the value 3.0.

ANINT (-2.798) has the value -3.0.

ANY

Transformational Intrinsic Function (Generic): Determines if *any* value is true in an entire array or in a specified dimension of an array.

Syntax

result = **ANY** (*mask* [, *dim*])

mask
(Input) Must be a logical array.

dim
(Optional; input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALL, COUNT

Example

```
    LOGICAL mask( 2, 3), AR1(3), AR2(2)
    DATA mask /T, T, F, T, F, F/
! mask is     true false false
!             true true false
    AR1 = ANY(mask,DIM = 1)  ! evaluates the elements column by
                             !   column yielding [true true false]
    AR2 = ANY(mask,DIM = 2)  ! evaluates the elements row by row
                             !   yielding [true true]
```

ANY ((/.FALSE., .FALSE., .TRUE./)) has the value true because one element is true.

A is the array

```
  [ 1  5  7 ]
  [ 3  6  8 ]
```

and B is the array

```
  [ 0  5  7 ]
  [ 2  6  9 ].
```

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.

APPENDMENUQQ

QuickWin Function: Appends a menu item to the end of a menu and registers its callback subroutine.

Module: USE DFLIB

Syntax

result = **APPENDMENUQQ** (*menuID, flags, text, routine*)

menuID
(Input) INTEGER(4). Identifies the menu to which the item is appended, starting with 1 as the leftmost menu.

flags
(Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive **OR** (see Results). The following constants are available:

- **$MENUGRAYED** - Disables and grays out the menu item.
- **$MENUDISABLED** - Disables but does not gray out the menu item.
- **$MENUENABLED** - Enables the menu item.
- **$MENUSEPARATOR** - Draws a separator bar.
- **$MENUCHECKED** - Puts a check by the menu item.
- **$MENUUNCHECKED** - Removes the check by the menu item.

text
(Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, 'WORDS OF TEXT'C.

routine
(Input) **EXTERNAL**. Callback subroutine that is called if the menu item is selected. All routines take a single **LOGICAL** parameter which indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- **WINPRINT** - Prints the program.
- **WINSAVE** - Saves the program.
- **WINEXIT** - Terminates the program.
- **WINSELTEXT** - Selects text from the current window.
- **WINSELGRAPH** - Selects graphics from the current window.
- **WINSELALL** - Selects the entire contents of the current window.
- **WINCOPY** - Copies the selected text and/or graphics from the current window to the Clipboard.
- **WINPASTE** - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a **READ.**
- **WINCLEARPASTE** - Clears the paste buffer.
- **WINSIZETOFIT** - Sizes output to fit window.
- **WINFULLSCREEN** - Displays output in full screen.
- **WINSTATE** - Toggles between pause and resume states of text output.
- **WINCASCADE** - Cascades active windows.
- **WINTILE** - Tiles active windows.
- **WINARRANGE** - Arranges icons.
- **WINSTATUS** - Enables a status bar.
- **WININDEX** - Displays the index for QuickWin help.
- **WINUSING** - Displays information on how to use Help.
- **WINABOUT** - Displays information about the current QuickWin application.
- **NUL** - No callback routine.

Results:

The result type is logical. It is .TRUE. if successful; otherwise, .FALSE..

You do not need to specify a menu item number, because **APPENDMENUQQ** always adds the new

item to the bottom of the menu list. If there is no item yet for a menu, your appended item is treated as the top-level menu item (shown on the menu bar), and *text* becomes the menu title. **APPENDMENUQQ** ignores the callback routine for a top-level menu item if there are any other menu items in the menu. In this case, you can set *routine* to NUL.

If you want to insert a menu item into a menu rather than append to the bottom of the menu list, use **INSERTMENUQQ**.

The constants available for flags can be combined with an inclusive OR where reasonable, for example **$MENUCHECKED** .OR. **$MENUENABLED**. Some combinations do not make sense, such as **$MENUENABLED** and **$MENUDISABLED**, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to **APPENDMENUQQ** as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

For more information about customizing QuickWin menus, see Using QuickWin in the *Programmer's Guide*.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: INSERTMENUQQ, DELETEMENUQQ, MODIFYMENUFLAGSQQ, MODIFYMENUROUTINEQQ, MODIFYMENUSTRINGQQ.

Example

```
      USE DFLIB
      LOGICAL(4) result
      CHARACTER(25) str
      ...
! Append two items to the bottom of the first (FILE) menu
      str    = '&Add to File Menu'C ! 'A' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
      str    = 'Menu Item &2b'C ! '2' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINCASCADE)
! Append an item to the bottom of the second (EDIT) menu
      str    = 'Add to Second &Menu'C ! 'M' is a quick-access key
      result = APPENDMENUQQ(2, $MENUENABLED, str, WINTILE)
```

ARC, ARC_W

Graphics Function: Draws elliptical arcs using the current graphics color.

Module: USE DFLIB

Syntax

result = **ARC** (*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)
result = **ARC_W** (*wx1*, *wy1*, *wx2*, *wy2*, *wx3*, *wy3*, *wx4*, *wy4*)

x1, *y1*
(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

x2, *y2*
(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

x3, *y3*
(Input) INTEGER(2). Viewport coordinates of start vector.

x4, *y4*
(Input) INTEGER(2). Viewport coordinates of end vector.

wx1,*wy1*
(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

wx3, *wy3*
(Input) REAL(8). Window coordinates of start vector.

wx4, *wy4*
(Input) REAL(8). Window coordinates of end vector.

Results:

The result type is INTEGER(2). It is nonzero if successful; otherwise, 0. If the arc is clipped or partially out of bounds, the arc is considered successfully drawn and the return is 1. If the arc is drawn completely out of bounds, the return is 0.

The center of the arc is the center of the bounding rectangle defined by the points (*x1*, *y1*) and (*x2*, *y2*) for **ARC** and (*wx1*, *wy1*) and (*wx2*, *wy2*) for **ARC_W**.

The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for **ARC** and (*wx3*, *wy3*) for **ARC_W**. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for **ARC** and (*wx4*, *wy4*) for **ARC_W**.

ARC uses the view-coordinate system. **ARC_W** uses the window-coordinate system. In each case, the arc is drawn using the current color.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

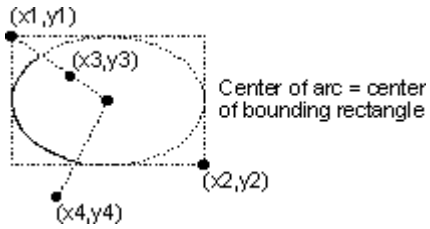
Example

This program draws the arc shown below.

```
USE DFLIB
INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4

x1 = 80; y1 = 50
x2 = 240; y2 = 150
x3 = 120; y3 = 75
x4 = 90; y4 = 180
status = ARC( x1, y1, x2, y2, x3, y3, x4, y4 )
END
```

Figure: Output of Program ARC.FOR



ASIN

Elemental Intrinsic Function (Generic): Produces an arcsine (with the result in radians).

Syntax

result = **ASIN** (*x*)

x
(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results:

The result type is the same as *x*. The value lies in the range -pi/2 to pi/2.

Specific Name	Argument Type	Result Type
ASIN	REAL(4)	REAL(4)
DASIN	REAL(8)	REAL(8)
QASIN [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ASIN (0.79345021) has the value 0.9164571.

ASIND

Elemental Intrinsic Function (Generic): Produces an arcsine (with the result in degrees).

Syntax

result = **ASIND** (*x*)

x
(Input) 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*.

Specific Name	Argument Type	Result Type
ASIND	REAL(4)	REAL(4)
DASIND	REAL(8)	REAL(8)
QASIND [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ASIND (0.2467590) has the value 14.28581.

ASM (Alpha only)

Nonelemental Intrinsic Function (Generic): Lets you use assembler instructions in an executable program.

Syntax

result = **ASM** (*string* [, a,...]})

string
Character. It is a character constant or a concatenation of character constants containing the assembler instructions.

a
(Optional) Any type. This can be a source or destination argument for the instruction, for

example.

Results:

The result is a scalar of type INTEGER(8), REAL(4), or REAL(8).

Arguments are passed by value. If you want to pass an argument by reference (for example, a whole array, a character string, or a record structure), you can use the %REF built-in function.

Labels are allowed, but all references must be from within the same **ASM** function. This lets you set up looping constructs, for example. Cross-jumping between **ASM** functions is not permitted.

In general, an **ASM** function can appear anywhere that an intrinsic function can be used. Since the supplied assembly code, assembly directives, or assembly data is integrated into the code stream, the compiler may choose to use different registers, better code sequences, and so on, just as if the code were written in Fortran.

You do not have absolute control over instruction sequences and registers, and the compiler may intersperse other code together with the **ASM** code for better performance. Better code sequences may be substituted by the optimizer if it chooses to do so.

Only register names beginning with a dollar sign ($) or percent sign (%) are permitted. For more information on register name conventions, see your operating system documentation set.

Specific Name	Argument Type [1]	Result Type
ASM [2]	CHARACTER	INTEGER(8)
FASM [3]	CHARACTER	REAL(4)
DASM [3]	CHARACTER	REAL(8)

[1] For the first argument.
[2] The value must be stored in register $0 by the user code.
[3] The value must be stored in register $F0 by the user code.

Examples

Consider the following:

```
! Concatenation is recommended for clarity.
! Notice that ";" separates instructions.
!
nine=9

type *, asm('addq %0, $17, $0;'//     ! Adds the first two arguments
    1    'ldq $22, %6;'//             !   and puts the answer in
    1    'ldq $23, %7;'//             !   register $0
    1    'ldq $24, %8;'//             !
    1    'mov $0, %fp;'//             ! Comments are not allowed in the
```

```
      1    'addq $18, %fp, $0;'//       !   constant, but are allowed here
      1    'addq $19, $0, $0;'//
      1    'addq $20, $0, $0;'//
      1    'addq $21, $0, $0;'//
      1    'addq $22, $0, $0;'//
      1    'addq $23, $0, $0;'//
      1    'addq $24, $0, $0;',
      1 1,2,3,4,5,6,7,8,nine)          ! The actual arguments to the
                                       !   ASM (usually by value)
 end
```

This example shows an integer **ASM** function that adds up 9 values and returns the sum as its result. Note that the user stores the function result in register $0.

All arguments are passed by value. The arguments not passed in registers can be named %6, %7, and %8, which correspond to the actual arguments 7, 8, and 9 (since %0 is the first argument). Notice that you can reference reserved registers like %fp.

The compiler creates the appropriate argument list. So, in this example, the first argument value (1) will be available in register $16, and the eighth argument value (8) will be available in %7, which is actually 8($30).

ASSIGN -- Label Assignment

Statement: Assigns a statement label value to an integer variable.

Syntax

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.

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.

The **ASSIGN** statement is an obsolescent feature of standard Fortran 90. Indirect branching through integer variables makes program flow difficult to read, especially if the integer variable is also used in arithmetic operations. Using these statements permits inconsistent usage of the integer variable, and can be an obscure source of error. The **ASSIGN** statement has been used to simulate internal procedures, which now can be coded directly.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Assignment: intrinsic, Obsolescent Features in Fortran 90

Examples

The value of a label is not the same as its number; instead, the label is identified by a number assigned by the compiler. In the following example, 400 is the label number (not the value) of IVBL:

```
ASSIGN 400 TO IVBL
```

Variables used in **ASSIGN** statements are not defined as integers. If you want to use a variable defined by an **ASSIGN** statement in an arithmetic expression, you must first define the variable by a computational assignment statement or by a **READ** statement, as in the following example:

```
IVBL = 400
```

The following example shows **ASSIGN** statements:

```
INTEGER ERROR
...
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
ASSIGN 250 TO ERROR
```

Note that NSTART and KSTOP are integer variables implicitly, but ERROR must be previously declared as an integer variable.

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.

Assignment(=) -- Defined Assignment

Statement: An interface block that defines 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:

Syntax

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 Unambiguous Generic Procedure References.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INTERFACE, Assignment Statements

Examples

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'
```

```
!Converting circle data to interval data.
module mod1
TYPE CIRCLE
        REAL radius, center_point(2)
END TYPE CIRCLE
TYPE INTERVAL
        REAL lower_bound, upper_bound
END TYPE INTERVAL
CONTAINS
        SUBROUTINE circle_to_interval(I,C)
          type (interval),INTENT(OUT)::I
          type (circle),INTENT(IN)::C
!Project circle center onto the x=-axis
!Note: the length of the interval is the diameter of the circle
                I%lower_bound = C%center_point(1) - C%radius
                I%upper_bound = C%center_point(1) + C%radius
        END SUBROUTINE circle_to_interval
end module mod1

PROGRAM assign
use mod1
TYPE(CIRCLE) circle1
TYPE(INTERVAL) interval1
INTERFACE ASSIGNMENT(=)
        module procedure circle_to_interval
END INTERFACE
!Begin executable part of program
  circle1%radius = 2.5
  circle1%center_point = (/3.0,5.0/)
  interval1 = circle1
. . .
END PROGRAM
```

Assignment -- Intrinsic

Statement: Assigns 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.

Syntax

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.

If the *c***DEC$ NOSTRICT** compiler directive (the default) is in effect, then you can assign a character expression to a noncharacter variable, and a noncharacter variable or array element (but not an expression) to a character variable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Assignment: defined, NOSTRICT directive

Example

```
    REAL a, b, c
    LOGICAL abigger
    CHARACTER(16) assertion
    c = .01
    a = SQRT (c)
    b = c**2

    assertion = 'a > b'
    abigger = (a .GT. b)

    WRITE (*, 100) a, b
100 FORMAT (' a =', F7.4, ' b =', F7.4)

    IF (abigger) THEN
            WRITE (*, *) assertion, ' is true.'
    ELSE
            WRITE (*, *) assertion, ' is false.'
    END IF
    END
```

```
! The program above has the following output:
!     a =    .1000     b =   .0001         a > b is true.

! The following code demonstrates legal and illegal
! assignment statements:
!
    INTEGER i, j
    REAL rone(4), rtwo(4), x, y
    COMPLEX z
    CHARACTER name6(6), name8(8)
    i         = 4
    x         = 2.0
    z         = (3.0, 4.0)
    rone(1) = 4.0
    rone(2) = 3.0
    rone(3) = 2.0
    rone(4) = 1.0
    name8   = 'Hello,'
! The following assignment statements are legal:
    i     = rone(2); j = rone(i); j = x
    y     = x; y = z; y = rone(3); rtwo = rone; rtwo = 4.7
    name6 = name8
! The following assignment statements are illegal:
    name6 = x + 1.0; int = name8//'test'; y = rone
    END
```

ASSOCIATED

Inquiry Intrinsic Function (Generic): Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.

Syntax

result = **ASSOCIATED** (*pointer* [, *target*])

pointer
(Input) Must be a pointer (of any data type).

target
(Optional; input) Must be a pointer or target. The pointer (in *pointer* or *target*) must not have an association status that is undefined.

Results:

The result type is default logical scalar.

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 option /integer_size can affect this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATED, POINTER, TARGET

Example

```
REAL C (:), D(:), E(5)
POINTER C, D
ARGET E
LOGICAL STATUS
C => E                        ! pointer assignment
D => E                        ! pointer assignment
STATUS = ASSOCIATED(C)        ! returns TRUE; C is associated
STATUS = ASSOCIATED(C, E)     ! returns TRUE; C is associated with E
STATUS = ASSOCIATED (C, D)    ! returns TRUE; C and D are associated
                              ! with the same target
```

Consider the following:

```
REAL, TARGET, DIMENSION (0:50) :: TAR
REAL, POINTER, DIMENSION (:) :: PTR
PTR => TAR
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 TAR has changed.

Consider the following:

```
REAL, POINTER, DIMENSION (:) :: PTR2, PTR3
ALLOCATE (PTR2 (0:15))
PTR3 => PTR2
ASSOCIATED (PTR2, PTR3)          ! Returns the value true
...
NULLIFY (PTR2)
NULLIFY (PTR3)
ASSOCIATED (PTR2, PTR3)          ! Returns the value false
```

ATAN

Elemental Intrinsic Function (Generic): Produces an arctangent (with the result in radians).

Syntax

result = **ATAN** (*x*)

x
(Input) Must be of type real.

Results:

The result type is the same as *x*. The value lies in the range -pi/2 to pi/2.

Specific Name	Argument Type	Result Type
ATAN	REAL(4)	REAL(4)
DATAN	REAL(8)	REAL(8)
QATAN [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ATAN (1.5874993) has the value 1.008666.

ATAND

Elemental Intrinsic Function (Generic): Produces an arctangent (with the result in degrees).

Syntax

result = **ATAND** (*x*)

x
(Input) Must be of type real and must be greater than or equal to zero.

Results:

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
ATAND	REAL(4)	REAL(4)
DATAND	REAL(8)	REAL(8)
QATAND [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Example

ATAND (0.0874679) has the value 4.998819.

ATAN2

Elemental Intrinsic Function (Generic): Produces an arctangent (with the result in radians). The result is the principal value of the argument of the nonzero complex number (x, y).

Syntax

result = **ATAN2** (*x*, *y*)

x
(Input) Must be of type real. It cannot have the value zero.

y
(Input) Must have the same type and kind parameters as *y*. It cannot have the value zero.

Results:

The result type is the same as *x*. The value lies in the range -pi to pi. 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 pi (if *x* < zero).

If *x* = zero, the absolute value of the result is pi/2.

Specific Name	Argument Type	Result Type
ATAN2	REAL(4)	REAL(4)
DATAN2	REAL(8)	REAL(8)
QATAN2 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y has the value

```
[  1   1 ]
[ -1  -1 ]
```

and X has the value

```
[ -1  1 ]
[ -1  1 ],
```

then ATAN2 (Y, X) is

$$\begin{bmatrix} \frac{3pi}{4} & \frac{pi}{4} \\ \frac{-3pi}{4} & \frac{-pi}{4} \end{bmatrix}$$

ATAN2D

Elemental Intrinsic Function (Generic): Produces an arctangent (with the result in degrees). The result is the principal value of the argument of the nonzero complex number (x, y).

Syntax

result = **ATAN2D** (*x*, *y*)

x
(Input) Must be of type real. It cannot have the value zero.

y
(Input) Must have the same type and kind parameters as *y*. It cannot have the value zero.

Results:

The result type is the same as *x*. 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.

Specific Name	Argument Type	Result Type
ATAN2D	REAL(4)	REAL(4)
DATAN2D	REAL(8)	REAL(8)
QATAN2D [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

ATAN2D (2.679676, 1.0) has the value 69.53546.

ATTRIBUTES

Compiler Directive: Declares properties for specified variables.

Syntax

*c***DEC$ ATTRIBUTES** *att* [, *att*]... **::** *object* [, *object*]...

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

att
Is one of the following:

ALIAS	DLLEXPORT	STDCALL
ARRAY_VISUALIZER	DLLIMPORT	VALUE
C	EXTERN	VARYING
DEFAULT	REFERENCE	

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
ALIAS	No	Yes	Yes
ARRAY_VISUALIZER[2]	Yes	No	No
C	No	Yes	Yes
DEFAULT	No	Yes	Yes
DLLEXPORT	No	Yes	Yes
DLLIMPORT	No	Yes	Yes
EXTERN	Yes	No	No
REFERENCE	Yes	No	Yes
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 arrays.

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:

- 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.

  The following example gives the subroutine happy the name OtherName outside this scoping unit.

```
INTERFACE
     SUBROUTINE happy
!DEC$ ATTRIBUTES C, VARYING, ALIAS:'OtherName' :: happy
     END SUBROUTINE
END INTERFACE
```

  c**DEC$ ATTRIBUTES** ALIAS has the same effect as the cDEC$ ALIAS directive.

- ARRAY_VISUALIZER

  Enhances the performance of the Array Visualizer.

  When declaring allocatable arrays to be viewed using the Array Viewer, this option can improve the performance of the Array Viewer. For example:

```
real(4), allocatable :: MyArray(:, :)
!DEC$ ATTRIBUTES array_visualizer :: MyArray
```

  When this option is used, array memory is shared between the Array Viewer and your application. Otherwise, the array data is copied during each faglUpdate call.

  This option is not useful unless the array is viewed in the Array Visualizer by using fagl* calls.

For more information on fagl* routines, see your online documentation for Array Visualizer.

- C and STDCALL

  Specify how data is to be passed when you use routines written in C or assembler with FORTRAN or Fortran 90 routines.

  On Intel processors, C and STDCALL have slightly different meanings; on all other platforms, they are interpreted as 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
Case of external subprogram names	VMS: Uppercase U*X: Lowercase WNT: Lowercase W95: Lowercase	VMS: Uppercase U*X: Lowercase WNT: Lowercase W95: Lowercase	VMS: Uppercase U*X: Lowercase WNT: Uppercase W95: Uppercase
U*X only:			
Trailing underscore added	No	No	Yes
WNT, W95:			
Leading underscore added	Yes	Yes	Yes
Number of arguments added	No	Yes	Yes
Caller stack cleanup	Yes	No	No
Variable number of arguments	Yes	No	No

[1] C and STDCALL are synonyms on OpenVMS and DIGITAL UNIX systems, and Windows NT systems on Alpha processors
[2] The Fortran 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 90 conventions pass arguments by

reference.

On Intel processors, an underscore (_) is placed at the beginning of the external name of a subprogram. If STDCALL is specified, an at sign (@) followed by the number of argument bytes being passed is placed at the end of the name. For example, a subprogram named SUB1 that has three INTEGER(4) arguments and is defined with STDCALL is assigned the external name _sub1@12.

Character arguments are passed as follows:

- By default:
  - On OpenVMS and DIGITAL UNIX Systems, hidden lengths are put at the end of the argument list.
  - On Windows NT and Windows 95 Systems, hidden lengths immediately follow the variable.
- If C or STDCALL (only) are specified:

  On all systems, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
- If C or STDCALL are specified with REFERENCE:

  On all systems, the string is passed with no length.

See also REFERENCE.

- 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 (/iface, /names, and /assume:underscore).

  This option can be combined with other *c***DEC$ 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.

- DLLEXPORT and DLLIMPORT (WNT, W95)

  Define a dynamic-link library's (DLL) interface for processes that use them. The properties can be assigned to data objects or procedures.

DLLEXPORT specifies that procedures or data are being exported to other applications or DLLs. This causes the compiler to produce efficient code, eliminating the need for a module definition (.def) file to export symbols.

If a procedure (or data) is declared with the DLLEXPORT property, it must be defined in the same module of the same program.

Symbols defined in a DLL are imported by programs that use them. The program must link with the import DLL and use the DLLIMPORT property inside the program unit that imports the symbol. DLLIMPORT is specified in a declaration, not a definition, since you cannot define a symbol you are importing.

For details on working with DLL applications, see Creating Fortran DLLs in the *Programmer's Guide*.

- 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.

- 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 following occurs:

  - On OpenVMS and DIGITAL UNIX systems, the string is passed with no length
  - On Windows NT and Windows 95 systems, hidden lengths immediately follow the variable

If REFERENCE and C (or STDCALL) are specified for a character argument, the string is passed with no length.

VALUE is the default if the C or STDCALL property is specified in the subprogram definition.

In the following example integer x is passed by value:

```
      SUBROUTINE Subr (x)
      INTEGER x
!DEC$ ATTRIBUTES VALUE :: x
```

- 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 these c**DEC$ ATTRIBUTES** options to individual arguments or to an entire routine.

The following form is also allowed: `!MS$ATTRIBUTES att [,att]... :: object [,object]...`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Programming with Mixed Languages, Creating Fortran DLLs, General Compiler Directives, Rules for General Directives

Examples

```
INTERFACE
   SUBROUTINE For_Sub (I)
     !DEC$ ATTRIBUTES C, ALIAS:'_For_Sub' :: For_Sub
     INTEGER I
   END SUBROUTINE For_Sub
END INTERFACE
```

You can assign more than one property to multiple variables with the same compiler directive. All properties apply to all the specified variables. For example:

```
 !DEC$ ATTRIBUTES REFERENCE, VARYING, C :: A, B, C
```

In this case, the variables `A`, `B`, and `C` are assigned the REFERENCE, VARYING, and C properties. The only restriction on the number of properties and variables is that the entire compiler directive

must fit on one line.

The identifier of the variable or procedure assigned properties must be a simple name. It cannot include initialization or array dimensions. For example, the following is not allowed:

```
 !DEC$ ATTRIBUTES C :: A(10)  ! This is illegal.
```

The following shows another example:

```
SUBROUTINE ARRAYTEST(arr)
!DEC$ ATTRIBUTES DLLEXPORT :: ARRAYTEST
   REAL(4) arr(3, 7)
   INTEGER i, j
   DO i = 1, 3
     DO j = 1, 7
       arr (i, j) = 11.0 * i + j
     END DO
   END DO
END SUBROUTINE
```

AUTOAddArg

DFAUTO Subroutine: Passes an argument name and value and adds the argument to the argument list data structure.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

CALL AUTOAddArg (*invoke_args*, *name*, *value* [, *output_arg*] [, *type*])

invoke_args
The argument list data structure of type INTEGER(4).

name
The argument's name of type CHARACTER*(*).

value
The argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types. Can also be of type VARIANT, which is defined in the DFCOMTY module.

output_arg
Indicates whether the argument's value is set by the called method. Must be of type LOGICAL. (See Note below.)

type
The variant type of the argument. Must be one of the VT_* constants defined in the DFCOMTY module.

Note: When the value of *output_arg* is TRUE, the variable used in the *value* parameter should be declared using the VOLATILE attribute. This is because the value of the variable will be changed by the subsequent call to AUTOInvoke. The compiler's global optimizations need to know that the value can change unexpectedly.

AUTOAllocateInvokeArgs

DFAUTO Function: Allocates an argument list data structure that holds the arguments to be passed to **AUTOInvoke**.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOAllocateInvokeArgs** ( )

Results:

The value returned is an argument list data structure of type INTEGER(4).

AUTODeallocateInvokeArgs

DFAUTO Subroutine: Deallocates an argument list data structure.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

CALL AUTODeallocateInvokeArgs (*invoke_args*)

invoke_args
The argument list data structure of type INTEGER(4).

AUTOGetExceptInfo

DFAUTO Subroutine: Retrieves the exception information when a method has returned an exception status.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

CALL AUTOGetExceptInfo (*invoke_args*, *code*, *source*, *description*, *help_file*, *help_context*, *scode*)

invoke_args

The argument list data structure of type INTEGER(4).

code
An output argument that returns the error code. Must be of type INTEGER(2).

source
An output argument that returns a human-readable name of the source of the exception. Must be of type CHARACTER*(*).

description
An output argument that returns a human-readable description of the error. Must be of type CHARACTER*(*).

help_file
An output argument that returns the fully qualified path of a Help file with more information about the error. Must be of type CHARACTER*(*).

help_context
An output argument that returns the Help context of the topic within the Help file. Must be of type INTEGER(4).

scode
An output argument that returns an SCODE describing the error. Must be of type INTEGER(4).

AUTOGetProperty

DFAUTO Function: Passes the name or identifier of the property and gets the value of the Automation object's property.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOGetProperty** (*idispatch*, *id*, *value* [, *type*])

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

id
The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value
An output argument that returns the argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type

The variant type of the requested argument. Must be one of the VT_* constants defined in the DFCOMTY module.

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyByID

DFAUTO Function: Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOGetPropertyByID** (*idispatch*, *memid*, *invoke_args*)

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

memid
Member ID of the property. Must be of type INTEGER(4).

invoke_args
The argument list data structure of type INTEGER(4).

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyInvokeArgs

DFAUTO Function: Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOGetPropertyInvokeArgs** (*idispatch*, *invoke_args*)

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

invoke_args
The argument list data structure of type INTEGER(4).

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOInvoke

DFAUTO Function: Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOInvoke** (*idispatch*, *id*, *invoke_args*)

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

id
The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

invoke_args
The argument list data structure of type INTEGER(4).

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOMATIC

Statement and Attribute: Controls the storage allocation of variables in subprograms (as does STATIC). Variables declared as AUTOMATIC and allocated in memory reside in the stack storage area, rather than at a static memory location.

The AUTOMATIC attribute can be specified in a type declaration statement or an **AUTOMATIC** statement, and takes one of the following forms:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **AUTOMATIC** [*att-ls*,] :: *v* [, *v*]...

Statement:

AUTOMATIC [::] *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 declarations only affect how data is allocated in storage.

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 in one of the following ways:

- As a keyword in a **FUNCTION** or **SUBROUTINE** statement
- As a compiler option
- As 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.

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 more than once in the same scoping unit.

If the variable is a pointer, AUTOMATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as AUTOMATIC. The following table shows these restrictions:

Variable	AUTOMATIC
Dummy argument	Yes
Automatic object	No
Common block item	No
Use-associated item	No
Function result	Yes
Component of a derived type	No

If a variable is in a module's outer scope, it*cannot* be specified as AUTOMATIC.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: STATIC, SAVE, Type declaration statements, Compatible attributes, RECURSIVE, /recursive, OPTIONS, POINTER, Modules and Module Procedures

Examples

The following example shows a type declaration statement specifying the AUTOMATIC attribute:

```
REAL, AUTOMATIC :: A, B, C
```

The following example uses an **AUTOMATIC** 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
...

C       In this example, all variables within the program unit
C       are automatic, except for "var1" and "var2"; these are
C       explicitly declared in a SAVE statement, and thus have
C       static memory locations:
        SUBROUTINE DoIt (arg1, arg2)

        INTEGER(4) arg1, arg2
```

```
        INTEGER(4) var1, var2, var3, var4

        AUTOMATIC
        SAVE var1, var3
C       var2 and var4 are automatic
```

AUTOSetProperty

DFAUTO Function: Passes the name or identifier of the property and a value, and sets the value of the Automation object's property.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOSetProperty** (*idispatch*, *id*, *value* [, *type*])

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

id
The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value
The argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type
The variant type of the argument. Must be one of the VT_* constants defined in the DFCOMTY module.

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyByID

DFAUTO Function: Passes the member ID of the property and sets the value of the Automation object's property into the argument list's first argument.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOSetPropertyByID** (*idispatch*, *memid*, *invoke_args*)

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

memid
Member ID of the property. Must be of type INTEGER(4).

invoke_args
The argument list data structure of type INTEGER(4).

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyInvokeArgs

DFAUTO Function: Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

Modules: USE DFAUTO, USE DFCOMTY

Syntax

result = **AUTOSetPropertyInvokeArgs** (*idispatch*, *invoke_args*)

idispatch
The object's IDispatch interface pointer. Must be of type INTEGER(4).

invoke_args
The argument list data structure of type INTEGER(4).

Results:

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

BACKSPACE

Statement: Positions a file at the beginning of the preceding record, making it available for subsequent I/O processing. It takes one of the following forms:

Syntax

BACKSPACE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])

BACKSPACE *io-unit*

io-unit
(Input) Is an external unit specifier.

label
Is the label of the branch target statement that receives control if an error occurs.

i-var
(Output) 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

Use the **BACKSPACE** statement with files connected for sequential access. **BACKSPACE** cannot be used to skip over records that have been written using list-directed or namelist formatting.

The I/O unit number must specify an open file on disk or magnetic tape.

Backspacing from the current record *n* is performed by rewinding to the start of the file and then performing *n*-1 successive **READ**s to reach the previous record.

A **BACKSPACE** statement must not be specified for a file that is open for direct or append access, because *n* is not available to the Fortran I/O system.

If a file is already positioned at the beginning of a file, a **BACKSPACE** statement has no effect.

If the file is positioned between the last record and the end-of-file record, **BACKSPACE** positions the file at the start of the last record.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REWIND, ENDFILE, Data Transfer I/O Statements, Branch Specifiers

Examples

```
BACKSPACE 5
BACKSPACE (5)
BACKSPACE lunit
BACKSPACE (UNIT = lunit, ERR = 30, IOSTAT = ios)
```

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.

BEEPQQ

Run-Time Subroutine: Sounds the speaker at the specified frequency for the specified duration in milliseconds.

Module: USE DFLIB

Syntax

CALL BEEPQQ (*frequency*, *duration*)

frequency
(Input) INTEGER(4). Frequency of the tone in Hz.

duration
(Input) INTEGER(4). Length of the beep in milliseconds.

BEEPQQ does not return until the sound terminates.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SLEEPQQ

Example

```
USE DFLIB
INTEGER(4) frequency, duration
frequency = 4000
duration  = 1000
CALL BEEPQQ(frequency, duration)
```

BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

Portability Functions: Compute the single-precision values of Bessel functions of the first and second kinds.

Module: USE DFPORT

Syntax

result = **BESJ0** (*posvalu*)
result = **BESJ1** (*posvalu*)
result = **BESJN** (*n, posvalu*)
result = **BESY0** (*posvalu*)
result = **BESY1** (*posvalu*)
result = **BESYN** (*n, posvalu*)

posvalue
(Input) REAL(4). Independent variable for a Bessel function. Must be greater than or equal to zero.

n
(Input) Default integer (INTEGER(4) unless changed by the user). Specifies the order of the selected Bessel function computation.

Results:

BESJ0, **BESJ1**, and **BESJN** return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

BESY0, **BESY1**, and **BESYN** return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause **BESY0**, **BESY1**, and **BESYN** to return QNAN.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DBESJ0, DBESJ1, DBESJN

BIC, BIS

Portability Subroutines: Perform a bit-level set and clear for integers.

Module: USE DFPORT

Syntax

CALL BIC (*bitnum, target*)
CALL BIS (*bitnum, target*)

bitnum
(Input) INTEGER(4). Bit number to set. Must be in the range 0 (least significant bit) to 31 (most significant bit).

target
(Input) INTEGER(4). Variable whose bit is to be set.

BIC sets bit *bitnum* of *target* to 0; **BIS** sets bit *bitnum* to 1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BIT

BIT

Portability Function: Performs a bit-level test for integers.

Module: USE DFPORT

Syntax

result = **BIT** (*bitnum, source*)

bitnum
(Input) INTEGER(4). Bit number to test. Must be in the range 0 (least significant bit) to 31 (most significant bit).

source
(Input) INTEGER(4). Variable being tested.

Results:

The result type is logical. .TRUE. if bit bitnum of source is 1; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BIC, BIS

BIT_SIZE

Inquiry Intrinsic Function (Generic): Returns the number of bits in an integer type.

Syntax

result = **BIT_SIZE** (*i*)

i
(Input) Must be of type default 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 Model for Bit Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IBCLR, IBITS, IBSET

Examples

BIT_SIZE (1_2) has the value 16 because the KIND=2 integer type contains 16 bits.

BLOCK DATA

Statement: Identifies a block-data program unit, which provides initial values for nonpointer variables in named common blocks.

Syntax

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	

[1] For more information, see RECORD statement and record structure declarations.
[2] Can only contain attributes: DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, or TARGET.
[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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: COMMON, DATA, EXTERNAL, Program Units and Procedures

Examples

The following shows 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
```

The following shows another example:

```
C      Main Program
       CHARACTER(LEN=10) LakeType
       REAL X(10), Y(4)
       COMMON/Lakes/a,b,c,d,e,family/Blk2/x,y
       ...
C      The following block-data subprogram initializes
C      the named common block /Lakes/:
C
       BLOCK DATA InitLakes
       COMMON /Lakes/ erie, huron, michigan, ontario,
      +        superior, fname
       DATA erie, huron, michigan, ontario, superior /1, 2, 3, 4, 5/
```

```
CHARACTER(LEN=10) fname/'GreatLakes'/
END
```

BSEARCHQQ

Run-Time Function: Performs a binary search of a sorted one-dimensional array for a specified element. The array elements cannot be derived types or structures.

Module: USE DFLIB

Syntax

result = **BSEARCHQQ** (*adrkey*, *adrarray*, *length*, *size*)

adrkey
(Input) INTEGER(4). Address of the variable containing the element to be found (returned by **LOC**).

adrarray
(Input) INTEGER(4). Address of the array (returned by **LOC**).

length
(Input) INTEGER(4). Number of elements in the array.

size
(Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory, specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$REAL4	REAL(4) or equivalent
SRT$REAL8	REAL(8) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

Results:

INTEGER(4). Array index of the matched entry, or 0 if the entry is not found.

The array must be sorted in ascending order before being searched.

Caution: The location of the array and the element to be found must both be passed by address using the **LOC** function. This defeats Fortran type checking, so you must make certain that the *length* and *size* arguments are correct, and that *size* is the same for the element to be found and the array searched.

If you pass invalid arguments, **BSEARCHQQ** attempts to search random parts of memory. If the memory it attempts to search is allocated to the current process, that memory is searched. If the memory it attempts to search is not allocated to the current process, the operating system intervenes, the program is halted, and you receive a General Protection Violation message.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SORTQQ, LOC

Example

```
USE DFLIB
INTEGER(4) array(10), length
INTEGER(4) result, target
length = SIZE(array)
...
result = BSEARCHQQ(LOC(target),LOC(array),length,SRT$INTEGER4)
```

BTEST

Elemental Intrinsic Function (Generic): Tests a bit of an integer argument.

Syntax

result = **BTEST** (*i*, *pos*)

i
(Input) Must be of type integer.

pos
(Input) Must be of type integer. It must not be negative and it must be less than **BIT_SIZE**(*i*).

Results:

The result type is 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 information on the model for the interpretation of an integer value as a sequence of bits, see Model for Bit Data.

The setting of compiler option /integer_size can affect this function.

Specific Name	Argument Type	Result Type
	INTEGER(1)	LOGICAL(1)
BITEST	INTEGER(2)	LOGICAL(2)
BTEST [1]	INTEGER(4)	LOGICAL(4)
BKTEST [2]	INTEGER(8)	LOGICAL(8)

[1] Or BJTEST
[2] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IBCLR, IBSET, IBCHNG, IOR, IEOR, IAND

Examples

BTEST (9, 3) has the value true.

If A has the value

```
[ 1  2 ]
[ 3  4 ],
```

the value of BTEST (A, 2) is

```
[ false  false ]
[ false   true ]
```

and the value of BTEST (2, A) is

```
[ true   false ]
[ false  false ].
```

The following shows more examples:

Function reference	*i*	Result
BTEST (*i*,2)	00011100 01111000	**.FALSE.**
BTEST (*i*,3)	00011100 01111000	**.TRUE.**

The following shows another example:

```
INTEGER(1) i(2)
LOGICAL result(2)
i(1) = 2#10101010
i(2) = 2#01010101
result = BTEST(i, (/3,2/))   ! returns (.TRUE.,.TRUE.)
write(*,*) result
```

BYTE

Statement: Specifies the **BYTE** data type, which is equivalent to INTEGER(1).

See Also: INTEGER, Integer Data Types

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
BYTE count, matrix(4, 4) / 4*1, 4*2, 4(4), 4*8 /
BYTE num  / 10 /
```

CALL

Statement: Transfers control to a subroutine subprogram.

Syntax

CALL *sub* [([*a-arg* [, *a-arg*]...])]

sub
Is the name of the subroutine 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 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 90 and Fortran 95.)

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 Functions Not Allowed as Actual Arguments).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SUBROUTINE, CONTAINS, RECURSIVE, USE, Program Units and Procedures

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.

The following shows another example:

```
!Variations on a subroutine call
      REAL S,T,X
      INTRINSIC NINT
      S=1.5
      T=2.5
      X=14.7
```

```
      !This calls SUB1 using keywords. NINT is an intrinsic function.
      CALL SUB1(B=X,C=S*T,FUNC=NINT,A=4.0)
!Here is the same call using an implicit reference
      CALL SUB1(4.0,X,S*T,NINT)
      CONTAINS
        SUBROUTINE sub1(a,b,c,func)
        INTEGER func
        REAL a,b,c
        PRINT *, a,b,c, func(b)
        END SUBROUTINE
      END
```

CASE

Statement: Marks the beginning of a **CASE** construct. A **CASE** construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a **SELECT CASE** statement.

Syntax

[*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:

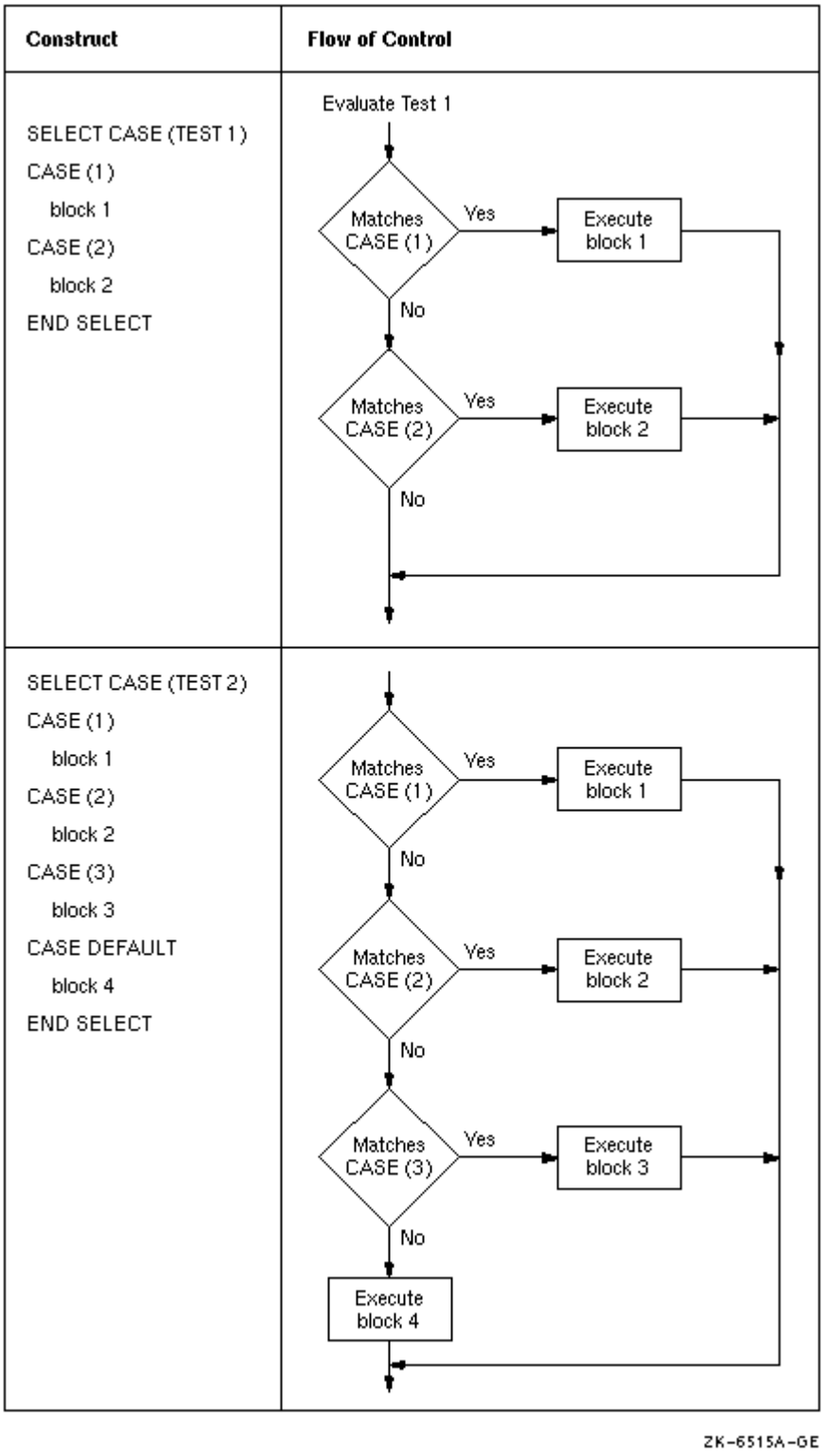
```
CASE (1, 4, 7, 11:14, 22)      ! Individual values as specified:
                               !     1, 4, 7, 11, 12, 13, 14, 22
CASE (:-1)                     ! All values less than zero
CASE (0)                       ! Only zero
CASE (1:)                      ! All values above zero
```

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.

The following figure shows the flow of control in a **CASE** construct:

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Execution Control

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
```

The following shows another example:

```
*CHARACTER*1 cmdchar
GET_ANSWER: SELECT CASE (cmdchar)
CASE ('0')
   WRITE (*, *) "Must retrieve one to nine files"
CASE ('1':'9')
   CALL RetrieveNumFiles (cmdchar)
CASE ('A', 'a')
   CALL AddEntry
CASE ('D', 'd')
   CALL DeleteEntry
CASE ('H', 'h')
   CALL Help
CASE DEFAULT
   WRITE (*, *) "Command not recognized; please use H for help"
END SELECT GET_ANSWER
```

CEILING

Elemental Intrinsic Function (Generic): Returns the smallest integer greater than or equal to its argument.

Syntax

result = **CEILING** (*a* [, *kind*])

a
(Input) Must be of type real.

kind
(Optional; input) Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results:

If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default integer. The value of the result is equal to the smallest integer greater than or equal to *a*. The result is undefined if the value cannot be represented in the default integer range.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FLOOR

Examples

CEILING (4.8) has the value 5.

CEILING (-2.55) has the value -2.0.

The following shows another example:

```
INTEGER I, IARRAY(2)
I = CEILING(8.01) ! returns 9
I = CEILING(-8.01) ! returns -8
IARRAY = CEILING((/8.01,-5.6/)) ! returns (9, -5)
```

CHANGEDIRQQ

Run-Time Function: Makes the specified directory the current, default directory.

Module: USE DFLIB

Syntax

result = **CHANGEDIRQQ (***dir***)**

dir
(Input) Character*(*). Directory to be made the current directory.

Results

LOGICAL(4). .TRUE. if successful; otherwise, .FALSE..

If you do not specify a drive in the *dir* string, the named directory on the current drive becomes the current directory. If you specify a drive in *dir*, the named directory on the specified drive becomes the current directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDRIVEDIRQQ, MAKEDIRQQ, DELDIRQQ, CHANGEDRIVEQQ

Example

```
USE DFLIB
LOGICAL(4) status
status = CHANGEDIRQQ('d:\fps90\bin\bessel')
```

CHANGEDRIVEQQ

Run-Time Function: Makes the specified drive the current, default drive.

Module: USE DFLIB

Syntax

result = **CHANGEDRIVEQQ** (*drive*)

drive
(Input) Character*(*). String beginning with the drive letter.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE.

Because drives are identified by a single alphabetic character, **CHANGEDRIVEQQ** examines only the first character of *drive*. The drive letter can be uppercase or lowercase.

CHANGEDRIVEQQ changes only the current drive. The current directory on the specified drive becomes the new current directory. If no current directory has been established on that drive, the root directory of the specified drive becomes the new current directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDRIVESQQ, GETDRIVESIZEQQ, GETDRIVEDIRQQ, CHANGEDIRQQ

Example

```
USE DFLIB
LOGICAL(4) status
status = CHANGEDRIVEQQ('d')
```

CHAR

Elemental Intrinsic Function (Generic): Returns the character in the specified position of the ASCII character set. It is the inverse of the function **ICHAR**.

Syntax

result = **CHAR** (*i* [, *kind*])

i
(Input) Must be of type integer with a value in the range 0 to n - 1, where *n* is the number of characters in the ASCII character set.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is character of length 1. The kind parameter is that of default character type.

The result is the character in position *i* of the ASCII 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) [2]	CHARACTER

[1] This specific function cannot be passed as an actual argument.
[2] INTEGER(8) is only available on Alpha processors.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ACHAR, IACHAR, ICHAR, ASCII and Key Code Charts

Examples

CHAR (76) has the value `'L'`.

CHAR (94) has the value `'^'`.

CHARACTER

Statement: Specifies the CHARACTER data type.

Syntax

CHARACTER
CHARACTER([KIND=]*n*)
CHARACTER*_len_

n
Is kind 1.

len
Is a string length (not a kind). For more information, see Declaration Statements for Character Types.

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

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Character Data Type, Character Constants, Character Substrings, C Strings, Declaration Statements for Character Types

Example

```
C
C   Length of wt and vs is 10, city is 80, and ch is 1
C
    CHARACTER wt*10, city*80, ch
    CHARACTER (LEN = 10), PRIVATE :: vs
    CHARACTER*(*) arg !declares a dummy argument
C   name and plume are ten-element character arrays
C   of length 20

    CHARACTER name(10)*20
    CHARACTER(len=20), dimension(10):: plume
C
C   Length of susan, patty, and dotty are 2, alice is 12,
C   jane is a 79-member array of length 2
C
    CHARACTER(2) susan, patty, alice*12, dotty, jane(79)
```

CHDIR

Portability Function: Changes the default directory.

Module: USE DFPORT

Syntax

result = **CHDIR** (*dir_name*)

dir_name
(Input) Character*(*). Name of a directory to become the default directory.

Results:

The result type is INTEGER(4). It returns zero if the directory was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The named directory does not exist.
- ENOTDIR: The *dir_name* parameter is not a directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CHANGEDIRQQ

Example

```
      integer(4) istatus, enoent, enotdir
      character*(*) newdir, prompt, errmsg
      prompt = 'Please enter directory name: '
10      write *, prompt
      read *, newdir
      ISTATUS = CHDIR(newdir)
      select case (istatus)
        case (enoent)
          errmsg = 'The directory'//newdir//' does not exist'
        case (enotdir)
          errmsg = newdir//' is not a directory'
        case else
          goto 40
      end select
      write *, errmsg
      goto 10
40      write *, 'Default directory successfully changed.'
      end
```

CHMOD

Portability Function: Changes the access mode of a file.

Module: USE DFPORT

Syntax

result = **CHMOD** (*name*, *mode*)

name
(Input) Character*(*). Name of the file whose access mode is to be changed. Must have a single path.

mode
(Input) Character*(*). File permission: either Read, Write, or Execute. The *mode* parameter can be either symbolic or absolute. An absolute mode is specified with an octal number, consisting of any combination of the following permission bits ORed together:

Permission bit	Description	Action
4000	Set user ID on execution	ignored; never true
2000	Set group ID on execution	ignored; never true
1000	Sticky bit	ignored; never true
0400	Read by owner	ignored; always true
0200	Write by owner	Settable

0100	Execute by owner	ignored; based on filename extension
0040, 0020, 0010	Read, Write, Execute by group	ignored; assumes owner permissions
0004, 0002, 0001	Read, Write, Execute by others	ignored; assumes owner permissions

The following regular expression represents a symbolic mode:

```
[ugoa]*[+-=] [rwxXst]*
```

"[ugoa]*" is ignored. "[+ - =]" indicates the operation to carry out:

- + Add the permission
- - Remove the permission
- = Absolutely set the permission

"[rwxXst]*" indicates the permission to add, subtract, or set. Only "w" is significant and affects write permission. All other letters are ignored.

Results:

INTEGER(4). Zero if the mode was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The specified file was not found.
- EINVAL: The mode argument is invalid.
- EPERM: Permission denied; the file’s mode cannot be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SETFILEACCESSQQ

Example

```
USE DFPORT
integer(4) I,Istatus
I = ACCESS ("DATAFILE.TXT", "w")
if (i) then
   ISTATUS = CHMOD ("datafile.txt", "[+w]")
end if
I = ACCESS ("DATAFILE.TXT","w")
print *, i
```

CLEARSCREEN

Graphics Subroutine: Erases the target area and fills it with the current background color.

Module: USE DFLIB

Syntax

CALL CLEARSCREEN (*area*)

area
(Input) INTEGER(4). Identifies the target area. Must be one of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GCLEARSCREEN** Clears the entire screen.
- **$GVIEWPORT** Clears only the current viewport.
- **$GWINDOW** Clears only the current text window (set with **SETTEXTWINDOW**).

All pixels in the target area are set to the color specified with **SETBKCOLORRGB**. The default color is black.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETBKCOLORRGB, SETBKCOLORRGB, SETTEXTWINDOW, SETVIEWPORT

Example

```
USE DFLIB
CALL CLEARSCREEN($GCLEARSCREEN)
```

CLICKMENUQQ

QuickWin Function: Simulates the effect of clicking or selecting a menu command. The QuickWin application responds as though the user had clicked or selected the command.

Modules: USE DFLIB

Syntax

result = **CLICKMENUQQ** (*item*)

item
(Input) INTEGER(4). Constant that represents the command selected from the Window menu. Must be one of the following symbolic constants (defined in DFLIB.F90 in the \DF98 \INCLUDE subdirectory):

- **QWIN$STATUS**: Status command
- **QWIN$TILE**: Tile command
- **QWIN$CASCADE**: Cascade command
- **QWIN$ARRANGE**: Arrange Icons command

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, REGISTERMOUSEEVENT, UNREGISTERMOUSEEVENT, WAITONMOUSEEVENT.

CLOCK

Portability Function: Converts a system time into an 8-character ASCII string.

Module: USE DFPORT

Syntax

result = **CLOCK** ( )

Results:

The result type is CHARACTER(8). The result is the current time in the form hh:mm:ss, using a 24-hour clock.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
USE DFPORT
character(8) whatimeisit
whatimeisit = CLOCK ()
write *, 'The current time is ',whatimeisit
```

CLOSE

Statement: Disconnects a file from a unit.

Syntax

CLOSE ([UNIT=]*io-unit* [, STATUS | DISPOSE | DISP = *p*] [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit
(Input) an external unit specifier.

p
(Input) 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 (unless **OPEN**(READONLY) is in effect).
- ’PRINT’ - Submits the file to the line print spooler, then retains it (sequential files only).
- ’PRINT/DELETE’ - Submits the file to the line print spooler, then deletes it (sequential files only).
- ’SUBMIT’ - Forks a process to execute the file.
- ’SUBMIT/DELETE’ - Forks a process to execute the file, then deletes the file after the fork is completed.

The default is ’DELETE’ for scratch files and QuickWin applications. For all other files, the default is ’KEEP’.

Files opened without a filename are called "scratch" files. Scratch files are temporary and are always deleted upon normal program termination; specifying STATUS=’KEEP’ for scratch files causes a run-time error.

For QuickWin applications, STATUS=’KEEP’ causes the child window to remain on the screen even after the unit closes. The default status is ’DELETE’, which removes the child window from the screen.

label
Is the label of the branch target statement that receives control if an error occurs.

i-var
(Output) 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= keyword 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.

You do not need to explicitly close open files. Normal program termination closes each file according to its default status. The **CLOSE** statement does not have to appear in the same program unit that opened the file.

Closing unit 0 automatically reconnects unit 0 to the keyboard and screen. Closing units 5 and 6 automatically reconnects those units to the keyboard or screen, respectively. Closing the asterisk (*****)

unit causes a compile-time error. In QuickWin, use **CLOSE** with unit 0, 5, or 6 to close the default window. If all of these units have been detached from the console (through an explicit **OPEN**), you must close one of these units beforehand to reestablish its connection with the console. You can then close the reconnect unit to close the default window.

If a parameter of the **CLOSE** statement is an expression that calls a function, that function must not cause an I/O operation or the **EOF** intrinsic function to be executed, because the results are unpredictable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Data Transfer I/O Statements, Branch Specifiers

Example

```
C     Close and discard file:
      CLOSE (7, STATUS = 'DELETE')
```

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.

CMPLX

Elemental Intrinsic Function (Generic): Converts the argument to complex type.

Syntax

result = **CMPLX** (*x* [, *y*] [, *kind*])

x
(Input) Must be of type integer, real, or complex.

y
(Optional; input) Must be of type integer or real. It must not be present if *x* is of type complex.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is 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 a complex value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, it is as if *y* were present with the value **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 option /real_size can affect this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DCMPLX, FLOAT, INT, IFIX, REAL, SNGL

Examples

CMPLX (-3) has the value (-3.0, 0.0).

CMPLX (4.1, 2.3) has the value (4.1, 2.3).

The following shows another example:

```
COMPLEX z1, z2
COMPLEX(8) z3
z1 = CMPLX(3)     ! returns the value 3.0 + i 0.0
z2 = CMPLX(3,4)   ! returns the value 3.0 + i 4.0
z3 = CMPLX(3,4,8) ! returns a COMPLEX(8) value 3.0D0 + i 4.0D0
```

COMAddObjectReference

DFCOM Function: Adds a reference to an object's interface.

Modules: USE DFCOM, USE DFCOMTY

Syntax

result = **COMAddObjectReference** (*iunknown*)

iunknown
An IUnKnown interface pointer. Must be of type INTEGER(4).

Results:

The result type is INTEGER(4). It is the object's current reference count.

For more information on the IUnKnown method **AddRef**, see the OLE section of the Win32 SDK.

COMCLSIDFromProgID

DFCOM Subroutine: Passes a programmatic identifier and returns the corresponding class identifier.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMCLSIDFromProgID (*prog_id*, *clsid*, *status*)

prog_id
The programmatic identifier of type CHARACTER*(*).

clsid
The class identifier corresponding to the programmatic identifier. Must be of type GUID, which is defined in the DFCOMTY module.

status
The status of the operation. It can be any status returned by **CLSIDFromProgID**. Must be of type INTEGER(4).

For more information on **CLSIDFromProgID**, see the OLE section of the Win32 SDK.

COMCLSIDFromString

DFCOM Subroutine: Passes a class identifier string and returns the corresponding class identifier.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMCLSIDFromString (*string*, *clsid*, *status*)

string
The class identifier string of type CHARACTER*(*).

clsid
The class identifier corresponding to the identifier string. Must be of type GUID, which is defined in the DFCOMTY module.

status
The status of the operation. It can be any status returned by **CLSIDFromString**. Must be of type INTEGER(4).

For more information on **CLSIDFromString**, see the OLE section of the Win32 SDK.

COMCreateObjectByGUID

DFCOM Subroutine: Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMCreateObjectByGUID (*clsid*, *clsctx*, *iid*, *interface*, *status*)

clsid
The class identifier of the class of object to be created. Must be of type GUID, which is defined in the DFCOMTY module.

clsctx
Lets you restrict the types of servers used for the object. Must be of type INTEGER(4). Must be one of the CLSCTX_* constants defined in the DFCOMTY module.

iid
The interface identifier of the interface being requested. Must be of type GUID, which is defined in the DFCOMTY module.

interface
An output argument that returns the object's interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by **CoCreateInstance**. Must be of type INTEGER(4).

For more information on **CoCreateInstance**, see the OLE section of the Win32 SDK.

COMCreateObjectByProgID

DFCOM Subroutine: Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMCreateObjectByProgID (*prog_id*, *idispatch*, *status*)

prog_id
The programmatic identifier of type CHARACTER*(*).

idispatch
An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by **CLSIDFromProgID** or **CoCreateInstance**. Must be of type INTEGER(4).

For more information on **CLSIDFromProgID** and **CoCreateInstance**, see the OLE section of the Win32 SDK.

COMGetActiveObjectByGUID

DFCOM Subroutine: Passes a class identifier and returns a pointer to the interface of a currently active object.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMGetActiveObjectByGUID (*clsid*, *clsctx*, *iid*, *interface*, *status*)

clsid
The class identifier of the class of object to be found. Must be of type GUID, which is defined in the DFCOMTY module.

clsctx
Lets you restrict the types of servers used for the object. Must be of type INTEGER(4). Must be one of the CLSCTX_* constants defined in the DFCOMTY module.

iid
The interface identifier of the interface being requested. Must be of type GUID, which is defined in the DFCOMTY module.

interface
An output argument that returns the object's interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by **CoGetClassObject**. Must be of type INTEGER(4).

For more information on **CoGetClassObject**, see the OLE section of the Win32 SDK.

COMGetActiveObjectByProgID

DFCOM Subroutine: Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMGetActiveObjectByProgID (*prog_id*, *idispatch*, *status*)

prog_id
The programmatic identifier of type CHARACTER*(*).

idispatch
An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by **CLSIDFromProgID** or **CoGetClassObject**. Must be of type INTEGER(4).

For more information on **CLSIDFromProgID** and **CoGetClassObject**, see the OLE section of the Win32 SDK.

COMGetFileObject

DFCOM Subroutine: Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMGetFileObject (*filename*, *idispatch*, *status*)

filename
The path of the file of type CHARACTER*(*).

idispatch
An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by **CreateBindCtx**, **MkParseDisplayName**, or the **IMonikerBindToObject** method. Must be of type INTEGER(4).

For more information on the **CreateBindCtx** or **MkParseDisplayName** routines or the **IMonikerBindToObject** method, see the OLE section of the Win32 SDK.

COMInitialize

DFCOM Subroutine: Initializes the COM library.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMInitialize (*status*)

status
The status of the operation. It can be any status returned by **OleInitialize**. Must be of type INTEGER(4).

You must use this routine to initialize the COM library before calling any other COM or AUTO routine.

For more information on **OleInitialize**, see the OLE section of the Win32 SDK.

COMMITQQ

Run-Time Function: Forces the operating system to execute any pending write operations for the file associated with a specified unit to the file's physical device.

Module: USE DFLIB

Syntax

result = **COMMITQQ** (*unit*)

unit
(Input) INTEGER(4). Fortran logical unit attached to a file to be flushed from cache memory to a physical device.

Results:

The result type is LOGICAL(4). If an open unit number is supplied, .TRUE. is returned and uncommitted records (if any) are written. If an unopened unit number is supplied, .FALSE. is returned.

Data written to files on physical devices is often initally written into operating-system buffers and then written to the device when the operating system is ready. Data in the buffer is automatically flushed to disk when the file is closed. However, if the program or the computer crashes before the data is transferred from buffers, the data can be lost. **COMMITQQ** tells the operating system to write any cached data intended for a file on a physical device to that device immediately. This is called flushing the file.

COMMITQQ is most useful when you want to be certain that no loss of data occurs at a critical point in your program; for example, after a long calculation has concluded and you have written the

results to a file, or after the user has entered a group of data items, or if you are on a network with more than one program sharing the same file. Flushing a file to disk provides the benefits of closing and reopening the file without the delay.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRINT, WRITE

Example

```
USE DFLIB
INTEGER unit / 10 /
INTEGER len
CHARACTER(80) stuff
OPEN(unit, FILE='COMMITQQ.TST', ACCESS='Sequential')
DO WHILE (.TRUE.)
  WRITE (*, '(A, \)') 'Enter some data (Hit RETURN to &
                       exit): '
  len = GETSTRQQ (stuff)
  IF (len .EQ. 0) EXIT
  WRITE (unit, *) stuff
  IF (.NOT. COMMITQQ(unit)) WRITE (*,*) 'Failed'
END DO
CLOSE (unit)
END
```

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*).

Syntax

COMMON [/[*cname*] /] *var-list* [[,] /[*cname*]/ *var-list*]...

cname
(Optional) 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. 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. Consider the following **COMMON** statements:

```
COMMON /ralph/ ed, norton, trixie
COMMON /     / fred, ethel, lucy
COMMON /ralph/ audrey, meadows
COMMON /jerry/ mortimer, tom, mickey
COMMON melvin, purvis
```

They are equivalent to these **COMMON** statements:

```
COMMON /ralph/ ed, norton, trixie, audrey, meadows
COMMON         fred, ethel, lucy, melvin, purvis
COMMON /jerry/ mortimer, tom, mickey
```

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.

Note: On DIGITAL UNIX, Windows NT, and Windows 95 systems, if a common block is initialized by a **DATA** statement, the module containing the initialization must declare the common block to be its maximum defined length.

This limitation does not apply if you compile all source modules together.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BLOCK DATA, DATA, MODULE, EQUIVALENCE, Specification expressions, Storage association, Interaction between COMMON and EQUIVALENCE Statements

Examples

```
PROGRAM MyProg
COMMON i, j, x, k(10)
COMMON /mycom/ a(3)
...
END
SUBROUTINE MySub
COMMON pe, mn, z, idum(10)
COMMON /mycom/ a(3)
...
END
```

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`	`SUBROUTINE FIGURE`
`...`	`COMMON /BLK1/LIMA,R / /ALFA,BET`
	`...`
`CALL FIGURE`	
`...`	`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)
```

The following example shows a valid association between subroutines in different program units. The object lists agree in number, type, and kind of data objects:

```
SUBROUTINE unit1
REAL(8)      x(5)
INTEGER      J
CHARACTER    str*12
TYPE(member) club(50)
COMMON / blocka / x, j, str, club
...

SUBROUTINE unit2
REAL(8)      z(5)
INTEGER      m
CHARACTER    chr*12
TYPE(member) myclub(50)
COMMON / blocka / z, m, chr, myclub
...
```

See also the program example for BLOCK DATA.

COMPLEX

Statement: Specifies the COMPLEX data type.

Syntax

COMPLEX
COMPLEX([KIND=]*n*)

COMPLEX*s*
DOUBLE COMPLEX

n
Is kind 4 or 8.

s
Is 8 or 16.**COMPLEX**(4) is specified as **COMPLEX***8. **COMPLEX**(8) is specified as **COMPLEX***16.

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**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DOUBLE COMPLEX, Complex Data Type, COMPLEX(4) Constants, COMPLEX(8) or DOUBLE COMPLEX Constants, Data Types, Constants, and Variables

Examples

```
COMPLEX ch
COMPLEX (KIND=4),PRIVATE :: zz, yy !equivalent to COMPLEX*8 zz, yy
COMPLEX(8) ax, by       !equivalent to COMPLEX*16 ax, by
COMPLEX (kind(4)) y(10)
complex (kind=8)  x, z(10)
```

COMQueryInterface

DFCOM Subroutine: Passes an interface identifier and returns a pointer to an object's interface.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMQueryInterface (*iunknown*, *iid*, *interface*, *status*)

iunknown
An IUnknown interface pointer. Must be of type INTEGER(4).

iid
The interface identifier of the interface being requested. Must be of type GUID, which is defined in the DFCOMTY module.

interface

An output argument that returns the object's interface pointer. Must be of type INTEGER(4).

status
The status of the operation. It can be any status returned by the IUnknown method **QueryInterface**. Must be of type INTEGER(4).

For more information on the IUnknown method **QueryInterface**, see the OLE section of the Win32 SDK.

COMReleaseObject

DFCOM Function: Indicates that the program is done with a reference to an object's interface.

Modules: USE DFCOM, USE DFCOMTY

Syntax

result = **COMReleaseObject** (*iunknown*)

iunknown
An IUnknown interface pointer. Must be of type INTEGER(4).

The result type is INTEGER(4). It is the object's current reference count.

COMUninitialize

DFCOM Subroutine: Uninitializes the COM library.

Modules: USE DFCOM, USE DFCOMTY

Syntax

CALL COMUninitialize ( )

When using COM routines, this must be the last routine called.

CONJG

Elemental Intrinsic Function (Generic): Calculates the conjugate of a complex number.

Syntax

result = **CONJG** (*z*)

z
(Input) 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)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: AIMAG

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).

The following shows another example:

```
COMPLEX z1
COMPLEX(8) z2
z1 = CONJG((3.0, 5.6))       ! returns (3.0, -5.6)
z2 = DCONJG((3.0D0, 5.6D0))  ! returns (3.0D0, -5.6D0)
```

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.

Syntax

CONTAINS

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Internal Procedures, Modules and Module Procedures, Main Program

Example

```
PROGRAM OUTER
   REAL, DIMENSION(10) :: A
```

```
   . . .
   CALL INNER (A)
CONTAINS
   SUBROUTINE INNER (B)
   REAL, DIMENSION(10) :: B
   . . .
   END SUBROUTINE INNER
END PROGRAM OUTER
```

CONTINUE

Statement: 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.

Syntax

CONTINUE

The statement by itself does nothing and has no effect on program results or execution sequence.

Compatibility

Console Standard Graphics QuickWin Graphics Windows DLL LIB

See Also: END DO, DO, Execution Control

Examples

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
```

The following shows another example:

```
     DIMENSION narray(10)
     DO 100 n = 1, 10
     narray(n) = 120
100  CONTINUE
```

COS

Elemental Intrinsic Function (Generic): Produces a cosine (with the result in radians).

Syntax

result = **COS** (*x*)

x
(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi. (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 [1]	REAL(16)	REAL(16)
CCOS [2]	COMPLEX(4)	COMPLEX(4)
CDCOS [3]	COMPLEX(8)	COMPLEX(8)

[1] VMS and U*X
[2] The setting of compiler option /real_size can affect CCOS.
[3] 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.

COSD

Elemental Intrinsic Function (Generic): Produces a cosine (with the result in degrees).

Syntax

result = **COSD** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

COSD (2.0) has the value 0.9993908.

COSD (30.4) has the value 0.8625137.

COSH

Elemental Intrinsic Function (Generic): Produces a hyperbolic cosine.

Syntax

result = **COSH** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

COSH (2.0) has the value 3.762196.

COSH (0.65893) has the value 1.225064.

COTAN

Elemental Intrinsic Function (Generic): Produces a cotangent (with the result in radians).

Syntax

result = **COTAN** (*x*)

x
(Input) Must be of type real; it cannot be zero. It must be in radians and is treated as modulo 2*pi.

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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

COTAN (2.0) has the value -4.576575E-01.

COTAN (0.6) has the value 1.461696.

COTAND

Elemental Intrinsic Function (Generic): Produces a cotangent (with the result in degrees).

Syntax

result = **COTAND** (*x*)

x
(Input) 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)
QCOTAND [1]	REAL(16)	REAL(16)
[1] VMS, U*X		

Examples

COTAND (2.0) has the value 0.2863625E+02.

COTAND (0.6) has the value 0.9548947E+02.

COUNT

Transformational Intrinsic Function (Generic): Counts the number of true elements in an entire array or in a specified dimension of an array.

Syntax

result = **COUNT** (*mask* [, *dim*])

mask
(Input) Must be a logical array.

dim
(Optional; input) 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 scalar of type default integer.

The result is 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALL, ANY

Example

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

```
[ 1  5  7 ]
[ 3  6  8 ]
```

and B is the array

```
[ 0  5  7 ]
[ 2  6  9 ].
```

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.

The following is another example:

```
LOGICAL mask (2, 3)
INTEGER AR1(3), AR2(2), I
mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., &
                 .FALSE., .FALSE./),(/2,3/))
! mask is the array   true false false
!                     true true false
AR1 = COUNT(mask,DIM=1) ! counts true elements by
                        ! column yielding [2 1 0]
AR2 = COUNT(mask,DIM=2) ! counts true elements by row
                        ! yielding [1 2]
I = COUNT( mask)        ! returns 3
```

CPU_TIME

Intrinsic Subroutine: Returns a processor-dependent approximation of the processor time in seconds. This is a Fortran 95 intrinsic subroutine.

Syntax

CALL CPU_TIME (*time*)

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_begin - time_end, ' seconds'
```

CSHIFT

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **CSHIFT** (*array*, *shift* [, *dim*])

array
(Input) Array whose elements are to be shifted. It can be of any data type.

shift
(Input) The number of positions shifted. 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

(Optional; input) Optional dimension along which to perform the shift. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EOSHIFT, ISHFT, ISHFTC

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 2 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 2 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

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ].
```

CSHIFT (M, SHIFT = 1, DIM = 2) produces the result

```
[ 2  3  1 ]
```

```
  [ 5  6  4 ]
  [ 8  9  7 ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 2 positions. The elements shifted off the beginning are inserted at the end.

CSHIFT (M, SHIFT = -1, DIM = 1) produces the result

```
  [ 7  8  9 ]
  [ 1  2  3 ]
  [ 4  5  6 ].
```

Each element in columns 1, 2, and 3 is shifted down by 1 position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, -1, 0/), DIM = 2) produces the result

```
  [ 2  3  1 ]
  [ 6  4  5 ]
  [ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 position; each element in row 2 is shifted to the *right* by 1 position; no element in row 3 is shifted at all.

The following is another example:

```
INTEGER array (3, 3), AR1(3, 3), AR2 (3, 3)
DATA array /1, 4, 7, 2, 5, 8, 3, 6, 9/
! array is   1 2 3
!            4 5 6
!            7 8 9
AR1 = CSHIFT(array, 1, DIM = 1)  ! shifts all columns
                                 ! by 1 yielding
                                 !         4 5 6
                                 !         7 8 9
                                 !         1 2 3
!
AR2=CSHIFT(array,shift=(/-1, 1, 0/),DIM=2) ! shifts
                                 ! each row separately
                                 ! by the amount in
                                 ! shift yielding
                                 !         3 1 2
                                 !         5 6 4
                                 !         7 8 9
```

CTIME

Portability Function: Converts a system time into a 24-character ASCII string.

Module: USE DFPORT

Syntax

result = **CTIME** (*stime*)

stime
(Input) INTEGER(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

Results:

The result is a value in the form Mon Jan 31 04:37:23 1994. Hours are expressed using a 24-hour clock.

The value of *stime* can be determined by calling the **TIME** function. **CTIME**(**TIME**( )) returns the current time and date.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
character (24) systime
systime = CTIME (TIME( ))
print *, 'Current date and time is ',systime
```

CYCLE

Statement: Interrupts the current execution cycle of the innermost (or named) **DO** construct.

Syntax

CYCLE [*name*]

name
(Optional) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DO, DO WHILE, DO Constructs

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
```

The following is from CYCLE.F90 in the /DF98/SAMPLES/TUTORIAL subdirectory:

```
      sample_loop: do i = 1, 5
           print *,i
             if( i .gt. 3 ) cycle sample_loop
           print *,i
           end do sample_loop
           print *,'done!'
!output:
!     1
!     1
!     2
!     2
!     3
!     3
!     4
!     5
!     done!
```

DATA

Statement: Assigns initial values to variables before program execution.

Syntax

DATA *var-list* /*clist*/ [[,] *var-list* /*clist*/]...

var-list
Is a list of variable names 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.

clist
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:

- A dummy argument
- A function
- A function result
- An automatic object

- An allocatable array
- A variable that is accessible by use or host association
- A variable in a named common block (unless the **DATA** statement is in a block data program unit)
- A 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.

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
```

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CHARACTER, INTEGER, REAL, COMPLEX, COMMON, Data Types, Constants, and Variables, I/O Lists

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.

The following shows another example:

```
    INTEGER n, order, alpha, list(100)
    REAL coef(4), eps(2),
    pi(5), x(5,5)
    CHARACTER*12 help
    COMPLEX*8 cstuff
    DATA  n /0/, order /3/
    DATA  alpha /'A'/
    DATA  coef /1.0, 2*3.0, 1.0/, eps(1) /.00001/
    DATA  cstuff /(-1.0, -1.0)/
!    The following example initializes diagonal and below in
!    a 5x5 matrix:
    DATA  ((x(j,i), i=1,j), j=1,5) / 15*1.0 /
    DATA  pi / 5*3.14159 /
    DATA  list / 100*0 /
    DATA  help(1:4), help(5:8), help(9:12) /3*'HELP'/
```

Consider the following:

```
 CHARACTER (LEN = 10) name
 INTEGER, DIMENSION (0:9) :: miles
 REAL, DIMENSION (100, 100) :: skew
 TYPE (member) myname, yours
 DATA name / 'JOHN DOE' /, miles / 10*0 /
 DATA ((skew (k, j), j = 1, k), k = 1, 100) / 5050*0.0 /
 DATA ((skew (k, j), j = k + 1, 100), k = 1, 99) / 4950*1.0 /
 DATA myname / member (21, 'JOHN SMITH') /
 DATA yours % age, yours % name / 35, 'FRED BROWN' /
```

In this example, the character variable `name` is initialized with the value JOHN DOE with two trailing blanks to fill out the declared length of the variable. The ten elements of `miles` are initialized to zero. The two-dimensional array `skew` is initialized so that its lower triangle is zero and its upper triangle is one. The structures `myname` and `yours` are declared using the derived type `member` from Derived Type. The derived-type variable `myname` is initialized by a structure constructor. The derived-type variable `yours` is initialized by supplying a separate value for each component.

The first **DATA** statement in the previous example could also be written as:

```
 DATA name / 'JOHN DOE' /
 DATA miles / 10*0 /
```

A pointer can be initialized as disassociated by using a **DATA** statement. For example:

```
INTEGER, POINTER :: P
DATA P/NULL()/
END
```

DATE

DATE can be used as an intrinsic subroutine or as a portability routine.

> **Warning:** The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

DATE Intrinsic Subroutine

Intrinsic Subroutine: Returns the current date as set within the system.

Syntax

> **CALL DATE** (*buf*)
>
> *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.

Example

```
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 1-character element, producing an incorrect result.

DATE Portability Routine

Portability Subroutine and Function: Returns the current system date.

Module: USE DFPORT

Subroutine Syntax

CALL DATE (*string*)

Function Syntax

result = **DATE** ( )

string
(Output) CHARACTER. Variable or array containing at least nine bytes of storage.

DATE in its function form returns a character(8) string in the form mm/dd/yy, where mm, dd, and yy are two-digit representations of the month, day, and year, respectively.

DATE in its subroutine form returns *string* in the form dd-mmm-yy, where dd is a two-digit representation of the current day of the month, mmm is a three-character abbreviation for the current month (for example, Jan) and yy are the last two digits of the current year.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFPORT
!If today's date is October 6, 1995, the following
!code prints "06-Oct-95"
CHARACTER(9) TODAY
CALL DATE(TODAY)
PRINT *, TODAY
!The next line prints "10/06/95"
PRINT *, DATE( )
```

DATE_AND_TIME

Intrinsic Subroutine: Returns character data on the real-time clock and date in a form compatible with the representations defined in Standard ISO 8601:1988.

Syntax

CALL DATE_AND_TIME ([*date*] [, *time*] [, *zone*] [, *values*])

date
(Optional; output) 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
(Optional; output) 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
(Optional; output) 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) in hours and parts of an hour expressed in minutes, respectively.

UTC (also known as Greenwich Mean Time) is defined by CCIR Recommendation 460-2.

values
(Optional; output) Must be of type default integer. One-dimensional array with size of at least 8. The values returned in *values* are as follows:

values (1) The 4-digit year

values (2) The month of the year

values (3) The day of the month

values (4) The time difference with respect to Coordinated Universal Time (UTC) in minutes

values (5) The hour of the day (range 0 to 23) - local time

values (6) The minutes of the hour (range 0 to 59) - local time

values (7) The seconds of the minute (range 0 to 59) - local time

values (8) The milliseconds of the second (range 0 to 999) - local time

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDAT, GETTIM, IDATE, FDATE, TIME, ITIME, RTC, CLOCK

Example

Consider the following example executed on 1993 April 23 at 13:23:30.5:

```
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 "19930423" to `REAL_CLOCK (1)`, the value "132330.500" to `REAL_CLOCK (2)`, and the value "+0100" to `REAL_CLOCK (3)`. The following values are assigned to `DATE_TIME`: 1993, 4, 23, 60, 13, 23, 30, and 500.

The following shows another example:

```
CHARACTER(10) t
CHARACTER(5) z
CALL DATE_AND_TIME(TIME = t, ZONE = z)
```

DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN

Portability Functions: Compute the double-precision values of Bessel functions of the first and second kinds.

Module: USE DFPORT

Syntax

result = **DBESJ0** (*posvalu*)
result = **DBESJ1** (*posvalu*)
result = **DBESJN** (*n, posvalu*)
result = **DBESY0** (*posvalu*)
result = **DBESY1** (*posvalu*)
result = **DBESYN** (*n, posvalu*)

posvalue
(Input) REAL(8). Independent variable for a Bessel function. Must be greater than or equal to zero.

n
(Input) Integer. Specifies the order of the selected Bessel function computation.

Results:

DBESJ0, **DBESJ1**, and **DBESJN** return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

DBESY0, **DBESY1**, and **DBESYN** return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause **DBESY0**, **DBESY1**, and **DBESYN** to return a huge negative value.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

Example

```
    USE DFPORT
    real(8) besnum, besout
10  read *, besnum
    besout = dbesj0(besnum)
    print *, 'result is ',besout
    goto 10
    end
```

DBLE

Elemental Intrinsic Function (Generic): Converts a number to double-precision real type.

Syntax

result = **DBLE** (*a*)

a
(Input) Must be of type integer, real, or complex.

Results:

The result type is 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)
DBLE [2]	REAL(4)	REAL(8)
	REAL(8)	REAL(8)
DBLEQ [3]	REAL(16)	REAL(8)
	COMPLEX(4)	REAL(8)
	COMPLEX(8)	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.
[3] VMS, U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FLOAT, SNGL, REAL, CMPLX

Examples

DBLE (4) has the value 4.0.

DBLE ((3.4, 2.0)) has the value 3.4.

DCMPLX

Elemental Intrinsic Function (Generic): Converts the argument to double complex type.

Syntax

result = **DCMPLX** (*x* [, *y*])

x
(Input) Must be of type integer, real, or complex.

y
(Optional; input) Must be of type integer or real. It must not be present if *x* is of type complex.

Results:

The result type is double complex (COMPLEX(8) or COMPLEX*16).

If only one noncomplex argument appears, it is converted into the real part of a complex value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, it is as if *y* were present with the value **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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CMPLX, FLOAT, INT, IFIX, REAL, SNGL

Examples

DCMPLX (-3) has the value (-3.0, 0.0).

DCMPLX (4.1, 2.3) has the value (4.1, 2.3).

DEALLOCATE

Statement: Frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated).

Syntax

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.

To disassociate a pointer that was not associated with the **ALLOCATE** statement, use the **NULLIFY** statement.

For a list of run-time errors, see Visual Fortran Run-Time Errors.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATE, NULLIFY, Arrays

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)
```

The following shows another example:

```
INTEGER, ALLOCATABLE :: dataset(:,:,:)
INTEGER reactor, level, points, error
DATA reactor, level, points / 10, 50, 10 /
ALLOCATE (dataset(1:reactor,1:level,1:points), STAT = error)
DEALLOCATE (dataset, STAT = error)
```

DECLARE and NODECLARE

Compiler Directives: DECLARE generates warnings for variables that have been used but have not been declared (like the **IMPLICIT NONE** statement). **NODECLARE** (the default) disables these warnings.

Syntax

*c***DEC$ DECLARE**
*c***DEC$ NODECLARE**

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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.

The following forms are also allowed: `!MS$DECLARE` and `!MS$NODECLARE`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IMPLICIT, General Compiler Directives

DECODE

Statement: Translates data from character to internal form. It is comparable to using internal files in formatted sequential **READ** statements.

Syntax

DECODE (c, f, b [, IOSTAT=i-var] [, ERR=label]) [io-list]

c
Is a scalar integer expression. It 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.
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.

label
Is the label of an executable statement that receives control if an error occurs.

io-list

Is an I/O list. An I/O list is either an implied-do list or a simple list of variables (except for assumed-size arrays).
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 **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.

See Also: READ, WRITE, ENCODE

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
```

DEFINE and UNDEFINE

Compiler Directives: DEFINE creates a symbolic variable whose existence or value can be tested during conditional compilation. **UNDEFINE** removes a defined symbol.

Syntax

*c***DEC$ DEFINE** *name* [= *val*]

*c***DEC$ UNDEFINE** *name*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

name
Is the name of the variable.

val
(Input) INTEGER(4). The value assigned to *name*.

Rules and Behavior

DEFINE and **UNDEFINE** create and remove symbols for use with the **IF** (or **IF DEFINED)** compiler directive. Symbols defined with **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.

The following forms are also allowed: `!MS$DEFINE name[=val]` and `!MS$UNDEFINE name`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IF Directive Construct, General Compiler Directives, /define compiler option

Example

```
!DEC$ DEFINE testflag
!DEC$ IF DEFINED (testflag)
   write (*,*) 'Compiling first line'
!DEC$ ELSE
   write (*,*) 'Compiling second line'
!DEC$ ENDIF
!DEC$ UNDEFINE testflag
```

DEFINE FILE

Statement: Establishes the size and structure of files with relative organization and associates them with a logical unit number.

Syntax

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*; *asv* must not be a dummy argument.

Rules and Behavior

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 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

```
DEFINE FILE 3(1000,48,U,NREC)
```

In this 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 forty-eight 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.

DELDIRQQ

Run-Time Function: Deletes a specified directory.

Module: USE DFLIB

Syntax

result = **DELDIRQQ** (*dir*)

dir
(Input) Character*(*). String containing the path of the directory to be deleted.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The directory to be deleted must be empty. It cannot be the current directory, the root directory, or a directory currently in use by another process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDRIVEDIRQQ, MAKEDIRQQ, CHANGEDIRQQ, CHANGEDRIVEQQ, UNLINK

Example

See the example for GETDRIVEDIRQQ.

DELETE

Statement: Deletes a record from a relative file.

Syntax

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 a relative file, 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 at that position.

> **Note**: You must use the /vms compiler option for **READ**s to detect that a record has been deleted.

See Also: Data Transfer I/O Statements, Branch Specifiers

Examples

The following statement deletes the fifth record in the file connected to I/O unit 10:

```
DELETE (10, REC=5)
```

Suppose the following statement is specified:

```
DELETE (UNIT=9, REC=10, IOSTAT=IOS, ERR=20)
```

The tenth record in the file connected to unit 9 is deleted. If an error occurs, control is transferred to the statement labeled 20, and a positive integer is stored in the variable IOS.

DELETEMENUQQ

QuickWin Function: Deletes a menu item from a QuickWin menu.

Module: USE DFLIB

Syntax

result = **DELETEMENUQQ (***menuID, itemID***)**

menuID
(Input) INTEGER(4). Identifies the menu that contains the menu item to be deleted, starting with 1 as the leftmost menu.

itemID
(Input) INTEGER(4). Identifies the menu item to be deleted, starting with 0 as the top menu item.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, APPENDMENUQQ, INSERTMENUQQ, MODIFYMENUFLAGSQQ, MODIFYMENUROUTINEQQ, MODIFYMENUSTRINGQQ.

Example

```
USE DFLIB
LOGICAL(4) result
CHARACTER(25) str
str = 'Add to EDIT Menu'C   ! Append to 2nd menu
result = APPENDMENUQQ(2, $MENUENABLED, str, WINSTATUS)
! Delete third item (EXIT) from menu 1 (FILE)
result = DELETEMENUQQ(1, 3)
! Delete entire fifth menu (WINDOW)
result = DELETEMENUQQ(5,0)
END
```

DELFILESQQ

Run-Time Function: Deletes all files matching the name specification, which can contain wildcards (* and ?).

Module: USE DFLIB

Syntax

result = **DELFILESQQ** (*files*)

files
(Input) Character*(*). File(s) to be deleted. Can contain wildcards (* and ?).

Results:

The result type is INTEGER(2). The result is the number of files deleted.

You can use wildcards to delete more than one file at a time. **DELFILESQQ** does not delete directories or system, hidden, or read-only files. Use this function with caution because it can delete many files at once. If a file is in use by another process (for example, if it is open in another process), it cannot be deleted.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FINDFILEQQ

Example

```
USE DFLIB
INTEGER(4) len, count
CHARACTER(80) file
CHARACTER(1) ch
WRITE(*,*) "Enter names of files to delete: "
len = GETSTRQQ(file)
IF (file(1:len) .EQ. '*.*') THEN
   WRITE(*,*) "Are you sure (Y/N)?"
   ch = GETCHARQQ()
   IF ((ch .NE. 'Y') .AND. (ch .NE. 'y')) STOP
END IF
count = DELFILESQQ(file)
WRITE(*,*) "Deleted ", count, " files."
END
```

Derived Type

Statement: Specifies the name of a user-defined type and the types of its components.

Syntax

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

access
Is the PUBLIC or PRIVATE 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 data 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.)

If **SEQUENCE** is present, 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. IIt 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.

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**( ).

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

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DIMENSION, MAP...END MAP, PRIVATE, PUBLIC, RECORD, SEQUENCE, STRUCTURE...END STRUCTURE, Derived Types, Default Initialization, Structure Components Structure Constructors

Examples

```
TYPE mem_name
  SEQUENCE
  CHARACTER (LEN = 20) lastn
  CHARACTER (LEN = 20) firstn
  CHARACTER (len = 3) cos   ! this works because COS is a component name
END TYPE mem_name
TYPE member
    TYPE (mem_name) :: name
    SEQUENCE
    INTEGER age
    CHARACTER (LEN = 20) specialty
END TYPE member
```

In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
TYPE (pair)
  INTEGER i, j
END TYPE
TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

The following example shows how you can use derived-type objects as components of other derived-type objects:

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
END TYPE
```

Objects of these derived types can then be used within a third derived-type specification, such as:

```
TYPE employee_data
  TYPE (employee_name) :: name
  TYPE (employee_addr) :: addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

%DESCR (VMS only)

Built-in Function: Changes the form of an actual argument. It passes an argument by descriptor.

Syntax

result = **%DESCR** (*a*)

a
(Input) An expression, record name, procedure name, array, character array section, or array element.

You must specify **%DESCR** in the actual argument list of a **CALL** statement or function reference. You cannot use it in any other context.

See Also: CALL, %VAL

Note: The following table lists the DIGITAL Fortran defaults for argument passing, and the allowed uses of **%DESCR**:

Actual Argument Data Type	Default	%DESCR
Expressions:		
Logical	REF	Yes
Integer	REF	Yes
REAL(4)	REF	Yes
REAL(8)	REF	Yes
REAL(16) [1]	REF	Yes
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
Character	DESCR [2]	Yes
Hollerith	REF	No
Aggregate [2]	REF	No
Derived	REF	No
Array Name:		
Numeric	REF	Yes
Character	DESCR [2]	Yes
Aggregate [3]	REF	No
Derived	REF	No
Procedure Name:		
Numeric	REF	Yes
Character	DESCR [2]	Yes

[1] VMS, U*X

[2] On DIGITAL UNIX, Windows NT and Windows 95 systems, a character argument is passed by address and hidden length.

[3] In DIGITAL Fortran record structures

DFLOAT

Elemental Intrinsic Function (Generic): Converts an integer to double precision type.

Syntax

result = **DFLOAT** (*a*)

a
(Input) Must be of type integer.

Results:

The result type is 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.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
DFLOTI	INTEGER(2)	REAL(8)
DFLOTJ	INTEGER(4)	REAL(8)
DFLOTK [2]	INTEGER(8)	REAL(8)

[1] These specific functions cannot be passed as actual arguments.
[2] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REAL

Examples

DFLOAT (-4) has the value -4.0.

DIGITS

Inquiry Intrinsic Function (Generic): Returns the number of significant digits for numbers of the same type and kind parameters as the argument.

Syntax

result = **DIGITS** (*x*)

x
(Input) 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 Model for Integer Data; real parameter *p* is defined in Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EXPONENT, RADIX, FRACTION, Data Representation Models

Examples

If *x* is of type REAL(4), DIGITS(x) has the value 24.

DIM

Elemental Intrinsic Function (Generic): Returns the difference between two numbers (if the difference is positive).

Syntax

result = **DIM** (*x*, *y*)

x
(Input) Must be of type integer or real.

y
(Input) 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 [2]	INTEGER(8)	INTEGER(8)
DIM	REAL(4)	REAL(4)
DDIM	REAL(8)	REAL(8)
QDIM [3]	REAL(16)	REAL(16)

[1] Or JIDIM.
[2] Alpha only
[3] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

DIM (6, 2) has the value 4.

DIM (-4.0, 3.0) has the value 0.0.

The following shows another example:

```
INTEGER i
REAL r
REAL(8) d
i = IDIM(10, 5)            ! returns 5
r = DIM (-5.1, 3.7)        ! returns 0.0
d = DDIM (10.0D0, -5.0D0)  ! returns 15.0D0
```

DIMENSION

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **DIMENSION** (*a-spec*) [, *att-ls*] **::** *a*[(*a-spec*)] [, *a*[(*a-spec*)]] ...

Statement:

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. It can be any of the following:

- An explicit-shape specification; for example, a(10,10)
- An assumed-shape specification; for example, a(**:**)
- A deferred-shape specification; for example, a(**:,:**)
- An assumed-size specification; for example, a(10,*)

For more information on array specifications, see Declaration Statements for Arrays.

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**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATE, Declaration Statements for Arrays, Arrays

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(8), DIMENSION(5,-2:2) :: A,B,C
```

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)
```

You can also declare arrays by using type and **ALLOCATABLE** statements, for example:

```
INTEGER A(2,0:2)
COMPLEX F
ALLOCATABLE F(:,:)
REAL(8), ALLOCATABLE, DIMENSION( :, :, : ) :: E
```

You can specify both the upper and lower dimension bounds. If, for example, one array contains data from experiments numbered 28 through 112, you could dimension the array as follows:

```
DIMENSION experiment(28:112)
```

Then, to refer to the data from experiment 72, you would reference experiment(72).

Array elements are stored in column-major order: the leftmost subscript is incremented first when the array is mapped into contiguous memory addresses. For example, consider the following statements:

```
INTEGER(2) a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

These are equivalent to:

```
INTEGER(2) a
DIMENSION a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

If a is placed at location 1000 in memory, the preceding **DATA** statement produces the following mapping.

Array element	Address	Value
a(1,0)	1000	1
a(2,0)	1002	2
a(1,1)	1004	3
a(2,1)	1006	4
a(1,2)	1008	5
a(2,2)	100A	6

The following **DIMENSION** statement defines an assumed-size array in a subprogram:

```
DIMENSION data (19,*)
```

At execution time, the array data is given the size of the corresponding array in the calling program.

The following program fragment dimensions two arrays:

```
 ...
 SUBROUTINE Subr (matrix, rows, vector)
 REAL MATRIX, VECTOR
 INTEGER ROWS
 DIMENSION MATRIX (ROWS,*), VECTOR (10),
+  LOCAL (2,4,8)
 MATRIX (1,1) = VECTOR (5)
 ...
```

DISPLAYCURSOR

Graphics Function: Controls cursor visibility.

Module: USE DFLIB

Syntax

result = **DISPLAYCURSOR** (*toggle*)

toggle
(Input) INTEGER(2). Constant that defines the cursor state. Has two possible values:

- **$GCURSOROFF**: Makes the cursor invisible regardless of its current shape and mode.
- **$GCURSORON**: Makes the cursor always visible in graphics mode.

Results:

The result type is INTEGER(2). The result is the previous value of *toggle*.

Cursor settings hold only for the currently active child window. You need to call DISPLAYCURSOR for each window in which you want the cursor to be visible.

A call to SETWINDOWCONFIG turns off the cursor.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

DLGEXIT

Run-Time Subroutine: Closes an open dialog box.

Module: USE DFLOGM

Syntax

CALL DLGEXIT (*dlg*)

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

If you want to exit a dialog box on a condition other than the user selecting the OK or Cancel button, you need to include a call to **DLGEXIT** from within your callback routine. **DLGEXIT** saves the data associated with the dialog box controls and then closes the dialog box. The dialog box is exited after **DLGEXIT** has returned control back to the dialog manager, not immediately after the call to **DLGEXIT**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGSETRETURN, DLGINIT, DLGMODAL, DLGMODELESS

Example

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE DFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
  CALL DLGEXIT (dlg)
```

DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR

Run-Time Functions: Retrieve the state of the dialog control variable.

Module: USE DFLOGM

Syntax

result = **DLGGET** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGGETINT** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGGETLOG** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGGETCHAR** (*dlg*, *controlid*, *value* [, *index*])

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

controlid
(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).

value
(Output) Integer, logical, or character. The value of the control's variable.

index
(Input; optional) Integer. Specifies the control variable whose value is retrieved. Necessary if the control has more than one variable of the same data type and you do not want to get the value of the default for that type.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the **DLGGET** functions to retrieve the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variable associated with it, but not necessarily all. The control variables are listed in Control Indexes in the *Programmer's Guide*. The types of controls they are associated with are listed in Available Indexes for Each Dialog Control.

You can use **DLGGET** to retrieve the value of any variable. You can also use **DLGGETINT** to retrieve an integer value, or **DLGGETLOG** and **DLGGETCHAR** to retrieve logical and character values, respectively. If you use **DLGGET**, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to retrieve a variable type that is not available, the **DLGGET** functions return .FALSE..

If two or more controls have the same *controlid*, you cannot use these controls in a **DLGGET** operation. In this case the function returns .FALSE..

The dialog box does not need to be open to access its control variables.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGSET, DLGSETSUB, DLGINIT, DLGMODAL, DLGMODELESS

Example

```
USE DFLOGM
INCLUDE "THISDLG.FD"
TYPE (DIALOG)  dlg
INTEGER        val
LOGICAL        retlog, is_checked
CHARACTER(256) text
...
retlog = DLGGET (dlg, IDC_CHECKBOX1, is_checked, dlg_status)
retlog = DLGGET (dlg, IDC_SCROLLBAR2, val, dlg_range)
retlog = DLGGET (dlg, IDC_STATIC_TEXT1, text, dlg_title)
...
```

DLGINIT, DLGINITWITHRESOURCEHANDLE

Run-Time Functions: Initialize a dialog box.

Module: USE DFLOGM

Syntax

result = **DLGINIT** (*id*, *dlg*)
result = **DLGINITWITHRESOURCEHANDLE** (*id*, *hinst*, *dlg*)

id
(Input) INTEGER(4). Dialog identifier. Can be either the symbolic name for the dialog or the identifier number, both listed in the Include file (with extension .FD).

dlg
(Output) Derived type DIALOG. Contains dialog box parameters.

hinst
(Input) INTEGER(4). Module instance handle in which the dialog resource can be found.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

DLGINIT must be called to initialize a dialog box before it can be used with **DLGMODAL**, **DLGMODELESS**, or any other dialog function.

DLGINIT will only search for the dialog box resource in the main application. For example, it will not find a dialog box resource that has been built into a dynamic link library.

DLGINITWITHRESOURCEHANDLE can be used when the dialog resource is not in the main application. If the dialog resource is in a dynamic link library (DLL),*hinst* must be the value passed as the first argument to the DLLMAIN procedure.

Dialogs can be used from any application, including console, QuickWin, and Windows.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGEXIT, DLGMODAL, DLGMODELESS, DLGUNINIT

Example

```
USE DFLOGM
INCLUDE 'DLG1.FD'
LOGICAL retlog
TYPE (DIALOG) thisdlg
...
retlog = DLGINIT (IDD_DLG3, thisdlg)
IF (.not. retlog) THEN
  WRITE (*,*) 'ERROR: dialog not found'
ELSE
...
```

DLGISDLGMESSAGE

Run-Time Function: Determines whether the specified message is intended for one of the currently displayed modeless dialog boxes.

Module: USE DFLOGM

Syntax

result = **DLGISDLSMESSAGE** (*mesg*)

mesg
(Input) Derived type T_MSG. Contains a Windows message.

Results:

The result type is LOGICAL(4). The result is .TRUE. if the message is processed by the dialog box. Otherwise, the result is .FALSE. and the message should be further processed.

DLGISDLSMESSAGE must be called in the message loop of Windows applications that display a modeless dialog box using **DLGMODELESS**. **DLGISDLSMESSAGE** determines whether the message is intended for one of the currently displayed modeless dialog boxes. If it is, it passes the message to the dialog box to be processed.

Compatibility

WINDOWS

See Also: DLGMODELESS, Using a Modeless Dialog Routine

Example

```
use dflogm
include 'resource.fd'
type (DIALOG)   dlg
type (T_MSG)     mesg
integer*4       ret
logical*4       lret
…
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
…
lret = DlgModeless(dlg, nCmdShow)
…
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
            lret = TranslateMessage( mesg )
            ret  = DispatchMessage( mesg )
   end if
end do
! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg.wParam
return
```

DLGMODAL

Run-Time Function: Displays a dialog box and processes user control selections made within the box.

Module: USE DFLOGM

Syntax

result = **DLGMODAL** (*dlg*)

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

Results:

The result type is INTEGER(4). By default, if successful, it returns the identifier of the control that caused the dialog to exit; otherwise, it returns -1. The return value can be changed with the

DLGSETRETURN subroutine.

During execution, **DLGMODAL** displays a dialog box and then waits for user control selections. When a control selection is made, the callback routine, if any, of the selected control (set with **DLGSETSUB)** is called.

The dialog remains active until an exit control is executed: either the default exit associated with the OK and Cancel buttons, or **DLGEXIT** within your own control callbacks. **DLGMODAL** does not return a value until the dialog box is exited.

The default return value for **DLGMODAL** is the identifier of the control that caused it to exit (for example, IDOK for the OK button and IDCANCEL for the Cancel button). You can specify your own return value with **DLGSETRETURN** from within one of your dialog control callback routines. You should not specify -1 as your return value, because this is the error value **DLGMODAL** returns if it cannot open the dialog.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGSETRETURN, DLGSETSUB, DLGINIT, DLGEXIT

Example

```
USE DFLOGM
INCLUDE "MYDLG.FD"
INTEGER return
TYPE (DIALOG) mydialog
...
return = DLGMODAL (mydialog)
...
```

DLGMODELESS

Run-Time Function: Displays a modeless dialog box.

Module: USE DFLOGM

Syntax

result = **DLGMODELESS** (*dlg* [, *nCmdShow*, *hwndParent*])

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user. The variable passed to this function must remain in memory for the duration of the dialog box, that is from the **DLGINIT** call through the **DLGUNINIT** call.

The variable can be declared as global data in a module, as a variable with the STATIC attribute, or in a calling procedure that is active for the duration of the dialog box. It must not

be an AUTOMATIC variable in the procedure that calls **DLGMODELESS**.

nCmdShow
(Input) Integer. Specifies how the dialog box is to be shown. It must be one of the following values:

Value	Description
SW_HIDE	Hides the dialog box.
SW_MINIMIZE	Minimizes the dialog box.
SW_RESTORE	Activates and displays the dialog box. If the dialog box is minimized or maximized, Windows restores it to its original size and position.
SW_SHOW	Activates the dialog box and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates the dialog box and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the dialog box and displays it as an icon.
SW_SHOWMINNOACTIVE	Displays the dialog box as an icon. The window that is currently active remains active.
SW_SHOWNA	Displays the dialog box in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays the dialog box in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays the dialog box. If the dialog box is minimized or maximized, Windows restores it to its original size and position.

The default value is SW_SHOWNORMAL.

hwndParent
(Input) Integer. Specifies the parent window for the dialog box. The default value is determined in this order:

1. If **DLGMODELESS** is called from a callback of a modeless dialog box, then that dialog box is the parent window.
2. The Windows desktop window is the parent window.

Results:

The result type is LOGICAL(4). The value is .TRUE. if the function successfully displays the dialog

box. Otherwise the result is .FALSE..

During execution, **DLGMODELESS** displays a modeless dialog box and returns control to the calling application. The dialog box remains active until **DLGEXIT** is called, either explicitly or as the result of the invocation of a default button callback.

DLGMODLESS can only be used in a Windows application. The application must contain a message loop that processes Windows messages. The message loop must call **DLGISDLGMESSAGE** for each message. See the example below. Multiple modeless dialog boxes can be displayed at the same time. A modal dialog box can be displayed from a modeless dialog box by calling **DLGMODAL** from a modeless dialog callback. However, **DLGMODELESS** cannot be called from a modal dialog box callback.

Use the DLG_INIT callback with **DLGSETSUB** to perform processing immediately after the dialog box is created and before it is displayed, and to perform processing immediately before the dialog box is destroyed.

Compatibility

WINDOWS

See Also: DLGSETSUB, DLGINIT, DLGEXIT, DLGISDLGMESSAGE, Using a Modeless Dialog Routine

Example

```
use dflogm
include 'resource.fd'
type (DIALOG)   dlg
type (T_MSG)     mesg
integer*4       ret
logical*4       lret
…
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
…
lret = DlgModeless(dlg, nCmdShow)
…
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
            lret = TranslateMessage( mesg )
            ret  = DispatchMessage( mesg )
   end if
end do
! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg.wParam
return
```

DLGSENDCTRLMESSAGE

Run-Time Function: Sends a Windows message to a dialog box control.

Module: USE DFLOGM

Syntax

result = **DLGSENDCTRLMESSAGE** (*dlg*, *controlid*, *msg*, *wparam*, *lparam*)

dlg
(Input) Derived-type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

controlid
(Input) Integer. Specifies the identifier of the control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).

msg
(Input) Integer. Derived type T_MSG. Specifies the message to be sent.

wparam
(Input) Integer. Specifies additional message specific information.

lparam
(Input) Integer. Specifies additional message specific information.

Results:

The result type is INTEGER(4). The value specifies the result of the message processing and depends upon the message sent.

The dialog box must be currently active by a call to **DLGMODAL** or **DLGMODELESS**. This function does not return until the message has been processed by the control.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGINIT, DLGSETSUB, DLGMODAL, DLGMODELESS

Example

```
use dfwin
use dflogm
include 'resource.fd'
type (dialog)    dlg
integer                 callbacktype
```

```
integer         cref
logical         lret

if (callbacktype == dlg_init) then
        ! Change the color of the Progress bar to red
        ! NOTE:  The following message succeeds only if Internet Explorer 4.0
        !        or later is installed
        cref = #FF              ! Red
        lret = DlgSendCtrlMessage(dlg, IDC_PROGRESS1, PBM_SETBARCOLOR, 0, cref)
endif
```

DLGSET, DLGSETINT, DLGSETLOG, DLGSETCHAR

Run-Time Functions: Set the values of dialog control variables.

Module: USE DFLOGM

Syntax

result = **DLGSET** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGSETINT** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGSETLOG** (*dlg*, *controlid*, *value* [, *index*])
result = **DLGSETCHAR** (*dlg*, *controlid*, *value* [, *index*])

dlg
(Input) Derived-type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

controlid
(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).

value
(Input) Integer, logical, or character. The value of the control's variable.

index
(Input; optional) Integer. Specifies the control variable whose value is set. Necessary if the control has more than one variable of the same data type and you do not want to set the value of the default for that type.

Results:

The result type is LOGICAL(4) The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the **DLGSET** functions to set the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variables associated with it, but not necessarily all. The control variables are listed in the table in Control Indexes in the *Programmer's Guide*. The types of controls they are associated with are listed in the table in Available Indexes for Each Dialog Control in the *Programmer's Guide*.

You can use **DLGSET** to set any control variable. You can also use **DLGSETINT** to set an integer variable, or **DLGSETLOG** and **DLGSETCHAR** to set logical and character values, respectively. If you use **DLGSET**, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to set a variable type that is not available, the **DLGSET** functions return .FALSE..

Calling **DLGSET** does not cause a callback routine to be called for the changing value of a control. In particular, when inside a callback, performing a **DLGSET** on a control does not cause the associated callback for that control to be called. Callbacks are invoked automatically only by user action on the controls in the dialog box. If the callback routine needs to be called, you can call it manually after the **DLGSET** is executed.

If two or more controls have the same *controlid*, you cannot use these controls in a **DLGSET** operation. In this case the function returns .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGSETSUB, DLGGET, Using Dialogs, Dialog Functions, and Dialog Controls

Example

```
USE DFLOGM
INCLUDE "DLGRADAR.FD"
TYPE (DIALOG)  dlg
LOGICAL        retlog
...
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 400, dlg_range)
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., dlg_status)
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, "Hot Button", dlg_title)
...
```

DLGSETRETURN

Run-Time Subroutine: Sets the return value for the **DLGMODAL** function from within a callback subroutine.

Module: USE DFLOGM

Syntax

CALL DLGSETRETURN (*dlg*, *retval*)

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

retval
(Input) Integer. Specifies the return value for **DLGMODAL** upon exiting.

DLGSETRETURN overrides the default return value with *retval*. You can set your own value as a means of determining the condition under which the dialog box was closed. The default return value for an error condition is -1, so you should not use -1 as your return value.

DLGSETRETURN should be called from within a callback routine, and is generally used with **DLGEXIT**, which causes the dialog box to be exited from a control callback rather than the user selecting the OK or Cancel button.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGEXIT, DLGMODAL

Example

```
SUBROUTINE SETRETSUB (dlg, button_id, callbacktype)
USE DFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG) dlg
LOGICAL     is_checked, retlog
INTEGER     return, button_id, callbacktype
...
retlog = DLGGET(dlg, IDC_CHECKBOX4, is_checked, dlg_state)
IF (is_checked) THEN
   return = 999
ELSE
   return = -999
END IF
CALL DLGSETRETURN (dlg, return)
CALL DLGEXIT (dlg)
END SUBROUTINE SETRETSUB
```

DLGSETSUB

Run-Time Function: Assigns your own callback subroutines to dialog controls and to the dialog box.

Module: USE DFLOGM

Syntax

result = **DLGSETSUB** (*dlg*, *controlid*, *value* [, *index*])

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

controlid
(Input) Integer. Specifies the identifier of a control within the dialog box. Can be the symbolic name for the control or the identifier number, both listed in the include (with extension .FD) file, or it can be the identifier of the dialog box.

value
(Input) EXTERNAL. Name of the routine to be called when the callback event occurs.

index
(Input; optional) Integer. Specifies which callback routine is executed when the callback event occurs. Necessary if the control has more than one callback routine.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

When a callback event occurs (for example, when you select a check box), the callback routine associated with that callback event is called. You use **DLGSETSUB** to specify the subroutine to be called. All callback routines should have the following interface:

SUBROUTINE *callbackname* (*dlg*, *control_id*, *callbacktype*)

callbackname
Is the name of the callback routine.

dlg
Refers to the dialog box and allows the callback to change values of the dialog controls.

control_id
Is the name of the control that caused the callback.

callbacktype
Indicates what callback is occurring (for example, DLG_CLICKED, DLG_CHANGE, or DLG_DBLCLICK).

The *control_id* and *callbacktype* parameters let you write a single subroutine that can be used with multiple callbacks from more than one control. Typically, you do this for controls comprising a logical group. You can also associate more than one callback routine with the same control, but you must use then use *index* parameter to indicate which callback routine to use.

The *control_id* can also be the identifier of the dialog box. The dialog box supports a single *callbacktype*, DLG_INIT. This callback is executed immediately after the dialog box is created with *callbacktype* DLG_INIT, and immediately before the dialog box is destroyed with *callbacktype* DLG_DESTROY.

Callback routines for a control are called after the value of the control has been updated based on the user’s action.

If two or more controls have the same *controlid*, you cannot use these controls in a **DLGSETSUB** operation. In this case, the function returns .FALSE..

For more information, see Dialog Callback Routines in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGSET, DLGGET

Example

```
PROGRAM DLGPROG
USE DFLOGM
INCLUDE "MYDLG.FD"
TYPE (dialog) mydialog
LOGICAL retlog
INTEGER return
EXTERNAL RADIOSUB
retlog = DLGINIT(IDD_mydlg, dlg)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON1, RADIOSUB)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON2, RADIOSUB)
return = DLGMODAL(dlg)
END
SUBROUTINE RADIOSUB( dlg, id, callbacktype )
  USE DFLOGM
  TYPE (dialog) dlg
  INTEGER id, callbacktype
  INCLUDE 'MYDLG.FD'
  CHARACTER(256) text
  INTEGER cel, far, retint
  LOGICAL retlog
  SELECT CASE (id)
    CASE (IDC_RADIO_BUTTON1)
    ! Radio button 1 selected by user so
    ! change text accordingly
      text = 'Statistics Package A'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
    CASE (IDC_RADIO_BUTTON2)
    ! Radio button 2 selected by user so
    ! change text accordingly
      text = 'Statistics Package B'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
  END SELECT
END SUBROUTINE RADIOSUB
```

DLGUNINIT

Run-Time Subroutine: Deallocates memory associated with an initialized dialog.

Module: USE DFLOGM

Syntax

CALL DLGUNINIT (*dlg*)

dlg
(Input) Derived type DIALOG. Contains dialog box parameters. The components of the type DIALOG are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

You should call **DLGUNINIT** when a dialog that was successfully initialized by **DLGINIT** is no longer needed. **DLGUNINIT** should only be called on a dialog initialized with **DLGINIT**. If it is called on an uninitialized dialog or one that has already been deallocated with **DLGUNINIT**, the result is undefined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DLGINIT, DLGMODAL, DLGMODELESS, DLGEXIT

Example

```
USE DFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG)  mydialog
LOGICAL        retlog
...
retlog = DLGINIT(IDD_mydlg, mydialog)
...
CALL DLGUNINIT (mydialog)
END
```

DO

Statement: Marks the beginning of a **DO** construct. The **DO** construct controls the repeated execution of a block of statements or constructs. (This repeated execution is called a *loop*.)

A **DO** construct takes one of the following forms:

Syntax

Block Form

[*name***:**] **DO** [*label*[,]] [*loop-control*]
    block
[*label*] *term-stmt*

Nonblock Form

DO *label*[,] [*loop-control*]

name
(Optional) Is the name of the **DO** construct.

label
(Optional) Is a statement label identifying the terminal statement.

loop-control
Is a DO iteration (see Iteration Loop Control) or a DO WHILE statement.

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 90 and Fortran 95.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CONTINUE, CYCLE, EXIT, DO WHILE, Execution Control, DO Constructs

Examples

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)
```

The following two program fragments are also examples of **DO** statements:

```
C   Initialize the even elements of a 20-element real array
C
    DIMENSION array(20)
    DO j = 2, 20, 2
      array(j) = 12.0
    END DO
C
C   Perform a function 11 times
C
    DO k = -30, -60, -3
      int = j / 3
      isb = -9 - k
      array(isb) = MyFunc (int)
    END DO
```

The following shows the final value of a DO variable (in this case 11):

```
  DO j = 1, 10
    WRITE (*, '(I5)') j
  END DO
  WRITE (*, '(I5)') j
```

DO WHILE

Statement: Executes the range of a **DO** construct while a specified condition remains true.

Syntax

DO [*label*[,]] **WHILE** (*expr*)

label
(Optional) Is a label specifying an executable statement in the same program unit.

expr
Is a scalar logical (test) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: CONTINUE, CYCLE, EXIT, DO, Execution Control, DO Constructs,

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)          DO 10 WHILE (I .GT. J)
  ARRAY(I,J) = 1.0             ARRAY(I,J) = 1.0
  I = I - 1                    I = I - 1
END DO                    10 END DO
```

The following shows another example:

```
CHARACTER(1) input
input = ' '
DO WHILE ((input .NE. 'n') .AND. (input .NE. 'y'))
   WRITE (*, '(A)') 'Enter y or n: '
   READ (*, '(A)') input
END DO
```

DOT_PRODUCT

Transformational Intrinsic Function (Generic): Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).

Syntax

result = **DOT_PRODUCT** (*vector_a*, *vector_b*)

vector_a
(Input) Must be a rank-one array of numeric (integer, real, or complex) or logical type.

vector_b
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRODUCT, MATMUL, TRANSPOSE

Examples

DOT_PRODUCT ((/1, 2, 3/), (/3, 4, 5/)) has the value 26 (calculated as follows: `((1 x 3) + (2 x 4) + (3 x 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.

The following shows another example:

```
I = DOT_PRODUCT((/1,2,3/), (/4,5,6/)) ! returns
                                      !  the value 32
```

DOUBLE COMPLEX

Statement: Specifies the **DOUBLE COMPLEX** data type.

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**(KIND=8) or **DOUBLE COMPLEX** constants. (See REAL and DOUBLE PRECISION for more information.)

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

For more information, see General Rules for Complex Constants.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: COMPLEX, Complex Data Types, DOUBLE PRECISION, REAL

Examples

```
DOUBLE COMPLEX vector, arrays(7,29)
DOUBLE COMPLEX pi, pi2 /3.141592654,6.283185308/
```

The following examples demonstrate valid and invalid **COMPLEX**(KIND=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 real constant.

DOUBLE PRECISION

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.

For more information, see General Rules for Real Constants.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REAL, REAL(8) or DOUBLE PRECISION Constants, Data Types, Constants, and Variables

Examples

```
DOUBLE PRECISION varnam
DOUBLE PRECISION,PRIVATE :: zz
```

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 type 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.

DPROD

Elemental Intrinsic Function (Specific): Produces a double precision product. This specific function has no generic function associated with it.

Syntax

result = **DPROD** (*x*, *y*)

x, *y*
(Input) Must be of type default real.

Results:

The result type is double precision real. The result value is equal to *x* * *y*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

DPROD (2.0, -4.0) has the value -8.00.

DPROD (5.0, 3.0) has the value 15.00.

The following shows another example:

```
REAL(8) d
d = DPROD (123456.7, 123456.7)
! returns 1.524155754649439E+010
```

DRAND, DRANDM

Portability Functions: Return double-precision random numbers in the range 0.0 through 1.0.

Module: USE DFPORT

Syntax

result = **DRAND** (*iflag*)
result = **DRANDM** (*iflag*)

iflag
(Input) INTEGER(4). Controls the way the random number is selected.

Results:

The result type is REAL(8). Return values are:

Value of *iflag*	**Selection process**
1	The generator is restarted and the first random value is selected.
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using *iflag*, restarted, and the first random value is selected.

There is no difference between **DRAND** and **DRANDM**. Both functions are included to insure portability of existing code that references one or both of them.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_NUMBER, RANDOM_SEED

Example

```
USE DFPORT
REAL(8) num
INTEGER(4) f
f=1
CALL print_rand
f=0
CALL print_rand
f=22
CALL print_rand
CONTAINS
SUBROUTINE print_rand
num = drand(f)
print *, 'f= ',f,':',num
END SUBROUTINE
```

```
END
```

DREAL

Elemental Intrinsic Function (Specific): Converts a double complex argument to double precision type. This specific function has no generic function associated with it.

DREAL must not be passed as an actual argument.

Syntax

result = **DREAL** (*a*)

a
(Input) Must be of type double complex (COMPLEX(8) or COMPLEX*16).

Results:

The result type is double precision real (REAL(8) or REAL*8).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REAL

Examples

DREAL ((2.0d0, 3.0d0)) has the value 2.00.

DTIME (WNT only)

Portability Function: Returns the elapsed CPU time since the start of program execution when first called, and the elapsed execution time since the last call to **DTIME** thereafter. This function is currently restricted to Windows NT systems.

Module: USE DFPORT

Syntax

result = **DTIME** (*tarray*)

tarray
(Output) REAL(4). Must be a rank one array with two elements:

- *tarray(1)* Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code.
- *tarray*(2) Elapsed system time, which is time spent executing privileged code (code in the

Windows Executive).

Results:

The result type is REAL(4). The result is the total CPU time, which is the sum of *tarray*(1) and *tarray* (2).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
 REAL(4) I, TA(2)
 I = DTIME(TA)
 write(*,*) 'Program has been running for', I, 'seconds.'
 write(*,*) ' This includes', TA(1), 'seconds of user time and', &
& TA(2), 'seconds of system time.'
```

ELEMENTAL

Keyword: Asserts that a user-defined procedure is a restricted form of pure procedure. This is a Fortran 95 feature.

To specify an elemental procedure, use the keyword in a **FUNCTION** or **SUBROUTINE** statement.

An elemental procedure can be passed an array, which is acted upon one element at a time.

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.

See Also: FUNCTION, SUBROUTINE

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
```

ELLIPSE, ELLIPSE_W

Graphics Function: Draws a circle or an ellipse using the current graphics color.

Module: USE DFLIB

Syntax

result = **ELLIPSE** (*control*, *x1*, *y1*, *x2*, *y2*)
result = **ELLIPSE_W** (*control*, *wx1*, *wy1*, *wx2*, *wy2*)

control
(Input) INTEGER(2). Fill flag. Can be one of the following symbolic constants:

- **$GFILLINTERIOR** Fills the figure using the current color and fill mask.
- **$GBORDER** Does not fill the figure.

x1, *y1*
(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

x2, *y2*
(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

wx1, *wy1*
(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

Results:

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the ellipse is clipped or partially out of bounds, the ellipse is considered successfully drawn, and the return is 1. If the ellipse is drawn completely out of bounds, the return is 0.

The border is drawn in the current color and line style.

When you use **ELLIPSE**, the center of the ellipse is the center of the bounding rectangle defined by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). When you use **ELLIPSE_W**, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The control option given by **$GFILLINTERIOR** is equivalent to a subsequent call to the **FLOODFILLRGB** function using the center of the ellipse as the start point and the current color (set by **SETCOLORRGB**) as the boundary color.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

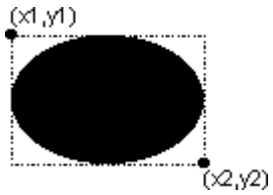
See Also: ARC, FLOODFILLRGB, GRSTATUS, LINETO, PIE, POLYGON, RECTANGLE, SETCOLORRGB, SETFILLMASK

Example

This program draws the shape shown below.

```
! compile as QuickWin or Standard Graphics application
 USE DFLIB
 INTEGER(2) dummy, x1, y1, x2, y2
 x1 = 80;  y1 = 50
 x2 = 240; y2 = 150
 dummy = ELLIPSE( $GFILLINTERIOR, x1, y1, x2, y2 )
 END
```

Figure: Output of Program ELLIPSE.FOR



ELSE Directive

See the IF Directive Construct.

ELSE

See IF Construct.

ELSEIF Directive

See the IF Directive Construct.

ELSE IF

See IF Construct.

ELSEWHERE

Statement: Marks the beginning of an **ELSEWHERE** block within a **WHERE** construct.

Syntax

[*name*:] **WHERE** (*mask-expr1*)
    [*where-body-stmt*]...
[**ELSEWHERE** (*mask-expr2*) [*name*]
    [*where-body-stmt*]...]
[**ELSEWHERE** [*name*]
    [*where-body-stmt*]...]
END WHERE [*name*]

name
Is the name of the **WHERE** construct.

mask-expr1, *mask-expr2*
Are logical array expressions (called mask expressions).

where-body-stmt
Is one of the following:

- An assignment statement of the form: array variable = array expression.
- A **WHERE** statement or construct

Rules and Behavior

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*".

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: WHERE

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
ELSEWHERE
  raining = .TRUE.
END WHERE
```

The variables `temp`, `pressure`, and `raining` are all arrays.

ENCODE

Statement: Translates data from internal (binary) form to character form. It is comparable to using internal files in formatted sequential **WRITE** statements.

Syntax

ENCODE (c, f, b [, IOSTAT=i-var] [, ERR=label]) [io-list]

c
Is a scalar integer expression. It 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.
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.

label
Is the label of an executable statement that receives control if an error occurs.

io-list
Is an I/O list. An I/O list is either an implied-do list or a simple list of variables (except for assumed-size arrays).
The list contains the data to be translated to character 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** 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.

See Also: READ, WRITE, DECODE

Examples

Consider the following:

```
      DIMENSION K(3)
      CHARACTER*12 A,B
      DATA A/'123456789012'/
      ENCODE(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’
```

END

Statement: Marks the end of a program unit. It takes one of the following forms:

Syntax

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Program Units and Procedures, Branch Statements

Example

```
C   An END statement must be the last statement in a program
C   unit:
    PROGRAM MyProg
    WRITE (*, ’("Hello, world!")’)
    END
C
C   An example of a named subroutine
C
```

```
    SUBROUTINE EXT1 (X,Y,Z)
      Real, Dimension (100,100) :: X, Y, Z
    END SUBROUTINE EXT1
```

END DO

Statement: Marks the end of a **DO** or **DO WHILE** loop.

Syntax

END DO

Remarks

There must be a matching **END DO** statement for every **DO** or **DO WHILE** statement that does not contain a label reference.

An **END DO** statement can terminate only one **DO** or **DO WHILE** statement. If you name the **DO** or **DO WHILE** statement, the **END DO** statement can specify the same name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DO, DO WHILE, CONTINUE

Example

The following examples both produce the same output:

```
  DO ivar = 1, 10
    PRINT ivar
  END DO
  ivar = 0

do2: DO WHILE (ivar .LT. 10)
        ivar = ivar + 1
        PRINT ivar
     END DO do2
```

ENDIF Directive

See the IF Directive Construct.

END IF

See IF Construct.

ENDFILE

Statement: Writes an end-of-file record to a sequential file and positions the file after this record (the terminal point). It can have either of the following forms.

Syntax

ENDFILE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])
ENDFILE *io-unit*

io-unit
(Input) Is an external unit specifier.

label
Is the label of the branch target statement that receives control if an error occurs.

i-var
(Output) 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.

End-of-file records should not be written in files that are read by programs written in a language other than Fortran.

Note: If you use the /vms compiler and an **ENDFILE** is performed on a sequential unit, an actual one byte record containing a Ctrl/Z is written to the file. If this option is not specified, an internal **ENDFILE** flag is set and the file is truncated. The option does not affect **ENDFILE** on relative files; such files are truncated.

If a parameter of the **ENDFILE** statement is an expression that calls a function, that function must not cause an I/O statement or the **EOF** intrinsic function to be executed, because unpredictable results can occur.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BACKSPACE, REWIND, Data Transfer I/O Statements, Branch Specifiers

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`.

The following shows another example:

```
WRITE (6, *) x
ENDFILE 6
REWIND 6
READ (6, *) y
```

END FORALL

Statement: Marks the end of a **FORALL** construct. For more information, see FORALL.

END INTERFACE

Statement: Marks the end of an **INTERFACE** block. For more information, see INTERFACE.

END WHERE

Statement: Marks the end of a **WHERE** block. For more information, see WHERE.

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
ELSEWHERE
  raining = .TRUE.
END WHERE
```

Note that the variables temp, pressure, and raining are all arrays.

ENTRY

Statement: Provides multiple entry points within a subprogram. It is not executable and must precede any **CONTAINS** statement (if any) within the subprogram.

Syntax

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
(Optional) Is a dummy argument. The dummy argument can be an alternate return indicator (*) if the **ENTRY** statement is within a subroutine subprogram.

r-name
(Optional) 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

An external or module procedure can have one or more **ENTRY** statements. Internal procedures must not contain **ENTRY** statements.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Program Units and Procedures, ENTRY Statements in Function Subprograms, ENTRY Statements in Subroutine Subprograms

Example

```
C  This fragment writes a message indicating
C  whether num is positive or negative
   IF (num .GE. 0) THEN
     CALL Sign
   ELSE
     CALL Negative
   END IF
   ...
   END

SUBROUTINE Sign
   WRITE (*, *) 'It''s positive.'
   RETURN
   ENTRY Negative
   WRITE (*, *) 'It''s negative.'
   RETURN
END SUBROUTINE
```

EOF

Inquiry Intrinsic Function (Generic): Checks whether a file is at or beyond the end-of-file record.

Syntax

result = **EOF** (*a*)

a
(Input) 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 type is 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..

This specific function cannot be passed as an actual argument.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ENDFILE, BACKSPACE, REWIND

Example

```
!  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
```

EOSHIFT

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **EOSHIFT** (*array*, *shift* [, *boundary*][, *dim*])

array
(Input) Must be an array (of any data type).

shift
(Input) 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
(Optional; input) 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). The *boundary* specifies a value to replace spaces left by the shifting procedure.

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
(Optional; input) 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CSHIFT, ISHFT, ISHFTC, TRANSPOSE

Examples

V is the array (1, 2, 3, 4, 5, 6).

EOSHIFT (V, SHIFT=2) shifts the elements in V to the *left* by 2 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

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ].
```

EOSHIFT (M, SHIFT = 1, BOUNDARY = ’*’, DIM = 2) produces the result

```
[ 2  3  * ]
[ 5  6  * ]
[ 8  9  * ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 1 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

```
[ 0  0  0 ]
[ 1  2  3 ]
[ 4  5  6 ].
```

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

```
[ 2  3  * ]
[ ?  4  5 ]
[ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 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.

The following is another example:

```
INTEGER shift(3)
CHARACTER(1) array(3, 3), AR1(3, 3)
array = RESHAPE ((/'A', 'D', 'G', 'B', 'E', 'H', &
                   'C', 'F', 'I'/), (/3,3/))
!        array is A B C
!                 D E F
!                 G H I
shift = (/-1, 1, 0/)
AR1 = EOSHIFT (array, shift, BOUNDARY = (/'*','?','#'/), DIM= 2)
! returns         * A B
!                 E F ?
!                 G H I
```

EPSILON

Inquiry Intrinsic Function (Generic): Returns a positive model number that is almost negligible compared to unity in the model representing real numbers.

Syntax

result = **EPSILON** (*x*)

x
(Input) Must be of type real; it can be scalar or array valued.

Results:

The result is 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 Model for Real Data.

EPSILON makes it easy to select a *delta* for algorithms (such as root locators) that search until the calculation is within *delta* of an estimate. If *delta* is too small (smaller than the decimal resolution of the data type), the algorithm might never halt. By scaling the value returned by **EPSILON** to the estimate, you obtain a *delta* that ensures search termination.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRECISION, TINY, Data Representation Models

Examples

If *x* is of type REAL(4), EPSILON (X) has the value 2^{-23}.

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.

Syntax

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:

- A dummy argument
- An allocatable array
- A pointer
- An object of nonsequence derived type
- An object of sequence derived type containing a pointer in the structure
- A function, entry, or result name
- A named constant
- A structure component
- A 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.

Visual Fortran lets you associate character and noncharacter entities, for example:

```
CHARACTER*1 char1(10)
REAL reala, realb
EQUIVALENCE (reala, char1(1))
EQUIVALENCE (realb, char1(2))
```

EQUIVALENCE statements require only the first subscript of a multidimensional array (unless the **STRICT** compiler directive is in effect). For example, the array declaration var(3,3), var(4) could appear in an **EQUIVALENCE** statement. The reference is to the fourth element of the array (var (1,2)), not to the beginning of the fourth row or column.

If you use the **STRICT** directive, the following rules apply to the kinds of variables and arrays that you can associate:

- If an **EQUIVALENCE** object is default integer, default real, double-precision real, default complex, default logical, or a sequenced derived type of all numeric or logical components, all objects in the **EQUIVALENCE** statement must be one of these types, though it is not necessary that they be the same type.
- If an **EQUIVALENCE** object is default character or a sequenced derived type of all character components, all objects in the **EQUIVALENCE** statement must be one of these types. The lengths do not need to be the same.
- If an **EQUIVALENCE** object is a sequenced derived type that is not purely numeric or purely character, all objects in the **EQUIVALENCE** statement must be the same derived type.
- If an **EQUIVALENCE** object is an intrinsic type other than the default (for example,

INTEGER(1)), all objects in the **EQUIVALENCE** statement must be the same type and kind.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EQUIVALENCE Statement, Initialization Expressions, Derived Data Types, Storage Association, STRICT Directive

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)
```

The following shows another example:

```
CHARACTER name, first, middle, last
DIMENSION name(60), first(20), middle(20), last(20)
EQUIVALENCE (name(1), first(1)), (name(21), middle(1))
EQUIVALENCE (name(41), last(1))
```

Consider the following:

```
CHARACTER (LEN = 4) :: a, b
```

```
CHARACTER (LEN = 3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

This causes the following alignment:

```
  1          2          3          4          5          6          7
a(1:1)     a(2:2)     a(3:3)     a(4:4)
                                 b(1:1)     b(2:2)     b(3:3)     b(4:4)
c(1)(1:1)  c(1)(2:2)  c(1)(3:3)  c(2)(1:1)  c(2)(2:2)  c(2)(3:3)
```

Note that the fourth element of a, the first element of b, and the first element of c(2) share the same storage unit.

ERRSNS

Intrinsic Subroutine: Returns information about the most recently detected I/O system error condition.

Syntax

ERRSNS ([io_err] [, sys_err] [, stat] [, unit] [, cond])

io_err
(Optional) Is an integer variable or array element that stores the most recent DIGITAL Fortran Run-Time Library error number that occurred during program execution. (For a listing of error numbers, see Visual Fortran Run-Time Errors.)

A zero indicates no error has occurred since the last call to **ERRSNS** or since the start of program execution.

sys_err
(Optional) Is an integer variable or array element that stores the most recent system error number associated with *io_err*. This code is the value returned by **GETLASTERROR**( ) at the time of the error.

stat
(Optional) Is an integer variable or array element that stores a status value that occurred during program execution. The value is zero.

unit
(Optional) Is an integer variable or array element that stores the logical unit number, if the last error was an I/O error.

cond
(Optional) 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)
```

ETIME (WNT only)

Portability Function: Returns the elapsed CPU time, in seconds, of the process that calls it. This function is currently restricted to Windows NT systems.

Module: USE DFPORT

Syntax

result = **ETIME** (*array*)

array
(Output) REAL(4). Must be a rank one array with two elements:

- *array(1)* Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code.
- *array(2)* Elapsed system time, which is time spent executing privileged code (code in the Windows Executive).

Results:

The result type is REAL(4). The result is the total CPU time, which is the sum of *array(1)* and *array (2)*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
REAL(4) I, TA(2)
I = ETIME(TA)
write(*,*) 'Program has used', I, 'seconds of CPU time.'
write(*,*) '  This includes', TA(1), 'seconds of user time and',  &
& TA(2), 'seconds of system time.'
```

EXIT

Statement: Terminates execution of a **DO** construct.

Syntax

EXIT [*name*]

name
(Optional) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DO, DO WHILE

Example

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
```

The following shows another example:

```
CC  See CYCLE.F90 in the /DF98/SAMPLES/TUTORIAL for an example of EXIT in nested
CC  DO loops
CC  Loop terminates early if one of the data points is zero:
CC
    INTEGER numpoints, point
    REAL datarray(1000), sum
    sum = 0.0
    DO point = 1, 1000
      sum = sum + datarray(point)
      IF (datarray(point+1) .EQ. 0.0) EXIT
    END DO
```

EXIT Subroutine

Intrinsic Subroutine: Terminates program execution, closes all files, and returns control to the operating system.

Syntax

CALL EXIT ([*status*])

status
(Optional; output) Is an integer argument you can use to specify the image exit-status value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: END, ABORT

Example

```
    INTEGER(4) exvalue
! all is well, exit with 1
    exvalue = 1
    CALL EXIT(exvalue)
! all is not well, exit with diagnostic -4
    exvalue = -4
    CALL EXIT(exvalue)
! give no diagnostic, just exit
    CALL EXIT ( )
```

EXP

Elemental Intrinsic Function (Generic): Computes an exponential value.

Syntax

result = **EXP** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
CEXP [2]	COMPLEX(4)	COMPLEX(4)
CDEXP [3]	COMPLEX(8)	COMPLEX(8)

[1] VMS and U*X
[2] The setting of compiler option /real_size can affect CEXP.
[3] This function can also be specified as ZEXP.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LOG

Examples

EXP (2.0) has the value 7.389056.

EXP (1.3) has the value 3.669297.

The following shows another example:

```
!  Given initial size and growth rate,
!  calculates the size of a colony at a given time.
     REAL sizei, sizef, time, rate
     sizei = 10000.0
     time = 40.5
     rate = 0.0875
     sizef = sizei * EXP (rate * time)
     WRITE (*, 100) sizef
100  FORMAT (' The final size is ', E12.6)
     END
```

EXPONENT

Elemental Intrinsic Function (Generic): Returns the exponent part of the argument when represented as a model number.

Syntax

result = **EXPONENT** (*x*)

x
(Input) must be of type real.

Results:

The result type is 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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: DIGITS, RADIX, FRACTION, MAXEXPONENT, MINEXPONENT, Data Representation Models

Examples

EXPONENT (2.0) has the value 2.

If 4.1 is a REAL(4) value, EXPONENT (4.1) has the value 3.

The following shows another example:

```
REAL(4) r1, r2
REAL(8) r3, r4
r1 = 1.0
r2 = 123456.7
r3 = 1.0D0
r4 = 123456789123456.7
write(*,*) EXPONENT(r1) ! prints 1
write(*,*) EXPONENT(r2) ! prints 17
write(*,*) EXPONENT(r3) ! prints 1
write(*,*) EXPONENT(r4) ! prints 47
END
```

EXTERNAL

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **EXTERNAL** [, *att-ls*] **::** *ex-pro* [, *ex-pro*]...

Statement:

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Program Units and Procedures, Type Declarations, INTRINSIC, Compatible attributes.

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)
```

The following shows another example:

```
EXTERNAL MyFunc, MySub
C   MyFunc and MySub are arguments to Calc
    CALL Calc (MyFunc, MySub)
C   Example of a user-defined function replacing an
C   intrinsic
    EXTERNAL SIN
    x = SIN (a, 4.2, 37)
```

FDATE

Portability Function and Subroutine: Returns the current date and time as an ASCII string.

Module: USE DFPORT

Subroutine Syntax

CALL FDATE ([*string*])

Function Syntax

result = **FDATE** ( )

string
(Optional; Output) Character*(*). When **FDATE** is called as a subroutine, *string* is returned as a 24-character string in the form:

```
Mon Jan 31 04:37:23 1996
```

Results:

The result of the function **FDATE** and the value of *string* returned by the subroutine **FDATE**(*string*) are identical. Newline and NULL are not included in the string.

When you use **FDATE** as a function, declare it as:

```
CHARACTER*24 FDATE
```

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
USE DFPORT
CHARACTER*24 today
!
CALL FDATE(today)
write (*,*), 'Today is ', today
!
write (*,*), 'Today is ', fdate()
```

FGETC

Portability Function: Reads the next available character from a file specified by a Fortran unit number.

Module: USE DFPORT

Syntax

result = **FGETC** (*lunit*, *char*)

lunit
(Input) INTEGER(4). Unit number of a file.

char
(Output) CHARACTER*1. Next available character in the file. If *lunit* is connected to a console device, then no characters are returned until the Enter key is pressed.

Results:

The result type is The result type is INTEGER(4). The result is zero if the read is successful, or -1 if an end-of-file is detected. A positive value is either a system error code or a Fortran I/O error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number).

If you use **WRITE**, **READ**, or any other Fortran I/O statements with *lunit*, be sure to read Input and Output With Portability Routines in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETCHARQQ, READ

Example

```
USE dfport
CHARACTER inchar
INTEGER istatus
istatus = FGETC(5,inchar)
PRINT *, inchar
END
```

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.

Syntax

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 Unit Specifier).

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 Record Specifier).

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 I/O Status Specifier).

See Also: Forms for Direct-Access READ Statements, I/O Control List

Example

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)
```

FINDFILEQQ

Run-Time Function: Searches for a specified file in the directories listed in the path contained in the environment variable.

Module: USE DFLIB

Syntax

result = **FINDFILEQQ** (*filename*, *varname*, *pathbuf*)

filename
(Input) Character*(*). Name of the file to be found.

varname
(Input) Character*(*). Name of an environment variable containing the path to be searched.

pathbuf
(Output) Character*(*). Buffer to receive the full path of the file found.

Results:

The result type is INTEGER(4). The result is the length of the string containing the full path of the found file returned in *pathbuf*, or 0 if no file is found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FULLPATHQQ, GETFILEINFOQQ, SPLITPATHQQ

Example

```
USE DFLIB
CHARACTER(256) pathname
INTEGER(4) pathlen
pathlen = FINDFILEQQ("libfmt.lib", "LIB", pathname)
WRITE (*,*) pathname
END
```

FIXEDFORMLINESIZE

Compiler Directive: Sets the line length for fixed-form Fortran source code.

Syntax

*c***DEC$ FIXEDFORMLINESIZE**:{72 | 80 | 132}

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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.

The following form is also allowed: `!MS$FIXEDFORMLINESIZE:{72|80|132}`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FREEFORM and NOFREEFORM, /fixed, Source Forms, General Compiler Directives

Example

```
cDEC$ NOFREEFORM
cDEC$ FIXEDFORMLINESIZE:132
WRITE (*,*) 'Sentence that goes beyond the 72nd column without continuation.'
```

FLOAT

Elemental Intrinsic Function: Converts an integer to REAL(4). For more information, see REAL.

FLOODFILL, FLOODFILL_W

Graphics Function: Fills an area using the current color index and fill mask.

Module: USE DFLIB

Syntax

result = **FLOODFILL** (*x*, *y*, *bcolor*)
result = **FLOODFILL_W** (*wx*, *wy*, *bcolor*)

x, *y*
(Input) INTEGER(2). Viewport coordinates for fill starting point.

wx, wy
(Input) REAL(8). Window coordinates for fill starting point.

bcolor
(Input) INTEGER(2). Color index of the boundary color.

Results:

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *bcolor*, or if the starting point lies outside the clipping region).

FLOODFILL begins filling at the viewport-coordinate point (*x*, *y*). **FLOODFILL_W** begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by **FLOODFILL** and **FLOODFILL_W** is set by **SETCOLOR.** You can obtain the current fill color index by calling **GETCOLOR**. These functions allow access only to the colors in the palette (256 or less). To access all available colors on a VGA (262,144 colors) or a true color system, use the RGB functions **FLOODFILLRGB** and **FLOODFILLRGB_W**.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current graphics color index set by **SETCOLOR**. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *bcolor*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: FLOODFILLRGB, FLOODFILLRGB_W, ELLIPSE, GETCOLOR, GETFILLMASK, GRSTATUS, PIE, SETCLIPRGN, SETCOLOR, SETFILLMASK

Example

```
USE DFLIB
INTEGER(2) status, bcolor, red, blue
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
x2 = 240; y2 = 150
red = 4
blue = 1
status = SETCOLOR(red)
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
bcolor = GETCOLOR()
status = SETCOLOR (blue)
xinterior = 160; yinterior = 100
status = FLOODFILL (xinterior, yinterior, bcolor)
END
```

FLOODFILLRGB, FLOODFILLRGB_W

Graphics Function: Fills an area using the current Red-Green-Blue (RGB) color and fill mask.

Module: USE DFLIB

Syntax

result = **FLOODFILLRGB** (*x*, *y*, *color*)
result = **FLOODFILLRGB_W** (*wx*, *wy*, *color*)

x, *y*
(Input) INTEGER(2). Viewport coordinates for fill starting point.

wx, *wy*
(Input) REAL(8). Window coordinates for fill starting point.

color
(Input) INTEGER(4). RGB value of the boundary color.

Results:

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *color*, or if the starting point lies outside the clipping region).

FLOODFILLRGB begins filling at the viewport-coordinate point (*x*, *y*). **FLOODFILLRGB_W** begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by **FLOODFILLRGB** and **FLOODFILLRGB_W** is set by **SETCOLORRGB.** You can obtain the current fill color by calling **GETCOLORRGB**.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current color set by **SETCOLORRGB**. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *color*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: ELLIPSE, FLOODFILL, GETCOLORRGB, GETFILLMASK, GRSTATUS, PIE, SETCLIPRGN, SETCOLORRGB, SETFILLMASK

Example

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(2) status
INTEGER(4) result, bcolor
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
x2 = 240; y2 = 150
result = SETCOLORRGB(#008080) ! red
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
```

```
bcolor = GETCOLORRGB( )
result = SETCOLORRGB (#FF0000) ! blue
xinterior = 160; yinterior = 100
result = FLOODFILLRGB (xinterior, yinterior, bcolor)
END
```

FLOOR

Elemental Intrinsic Function (Generic): Returns the greatest integer less than or equal to its argument.

Syntax

result = **FLOOR** (*a* [, *kind*])

a
(Input) Must be of type real.

kind
(Optional; input) Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results:

If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default integer. The result value is equal to the greatest integer less than or equal to *a*. The result is undefined if the value cannot be represented in the default integer range.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CEILING

Examples

FLOOR (4.8) has the value 4.

FLOOR (-5.6) has the value -6.

The following shows another example:

```
I = FLOOR(3.1)  ! returns 3
I = FLOOR(-3.1) ! returns -4
```

FLUSH

Portability Subroutine: Flushes the contents of an external unit buffer into its associated file.

Module: USE DFPORT

Syntax

CALL FLUSH (*lunit*)

lunit
(Input) INTEGER(4). Number of the external unit to be flushed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: COMMITQQ

FOCUSQQ

QuickWin Function: Sets focus to the window with the specified unit number.

Module: USE DFLIB

Syntax

result = **FOCUSQQ** (*iunit*)

unit
(Input) INTEGER(4). Unit number of the window to which the focus is set. Unit numbers 0, 5, and 6 refer to the default startup window.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

Units 0, 5, and 6 refer to the default window only if the program does not specifically open them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

Unlike **SETACTIVEQQ**, **FOCUSQQ** brings the specified unit to the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling **SETACTIVEQQ**.

A window has focus when it is given the focus by **FOCUSQQ**, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If

IOFOCUS=.TRUE., the child window receives focus prior to each **READ**, **WRITE**, **PRINT**, or **OUTTEXT**. Calls to graphics functions (such as **OUTGTEXT** and **ARC**) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, SETACTIVEQQ, INQFOCUSQQ.

FOR_CHECK_FLAWED_PENTIUM

Run-Time Function: Checks the processor to determine if it shows characteristics of the Pentium® floating-point divide flaw.

This routine can be called from a C program. It is invoked by default from a Fortran program unless /check:noflawed_pentium is specified.

Module: USE DFLIB

Syntax

result = **FOR_CHECK_FLAWED_PENTIUM** ( )

Results:

If the floating-point divide flaw is found, a severe forrtl error message is displayed and the calling program is terminated.

You can bypass this action by setting environment variable FOR_RUN_FLAWED_PENTIUM to the value TRUE.

For more information, see Intel Pentium Floating-Point Flaw.

Example

Consider the following C code:

```
void __stdcall for_check_flawed_pentium ( void );
for_check_flawed_pentium ( );
```

Consider the following Fortran code that checks for the divide flaw:

```
USE DFLIB

REAL*8 X, Y, Z

X = 5244795.0
Y = 3932159.0
```

```
Z = X - (X/Y) * Y
IF (Z .NE. 0) THEN           ! If flawed, Z will be 256
  PRINT *, " FDIV flaw detected on Pentium"
ENDIF
```

FOR_GET_FPE

Run-Time Function: Returns the current settings of floating-point exception flags. This routine can be called from a C or Fortran program.

Module: USE DFLIB

Syntax

result = **FOR_GET_FPE** ( )

Results:

The result type is INTEGER(4). The return value represents the settings of the current floating-point exception flags. The meanings of the bits are defined in the DFLIB module file.

To set floating-point exception flags after program initialization, use FOR_SET_FPE.

Example

Consider the following:

```
USE DFLIB

INTEGER*4 FOR_GET_FPE, FPE_FLAGS
EXTERNAL FOR_GET_FPE
FPE_FLAGS = FOR_GET_FPE ( )
```

FOR_RTL_FINISH_

Run-Time Function: Cleans up the Fortran run-time environment; for example, flushing buffers and closing files. It also issues messages about floating-point exceptions, if any occur.

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

Syntax

result = **FOR_RTL_FINISH_** ( )

Results:

The result is an I/O status value. For information on these status values, see Using the IOSTAT Value and Fortran Exit Codes.

To initialize the Fortran run-time environment, use FOR_RTL_INIT_.

Example

Consider the following C code:

```
int io_status;
int for_rtl_finish_ ( );
io_status = for_rtl_finish_ ( );
```

FOR_RTL_INIT_

Run-Time Subroutine: Initializes the Fortran run-time environment. It establishes handlers and floating-point exception handling, so Fortran subroutines behave the same as when called from a Fortran main program.

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

Syntax

CALL FOR_RTL_INIT_ (*argcount*, *actarg*)

argcount
Is a command-line parameter describing the argument count.

actarg
Is a command-line parameter describing the actual arguments.

To clean up the Fortran run-time environment, use FOR_RTL_FINISH_.

Example

Consider the following C code:

```
int argc;
char **argv;
void for_rtl_init_ (int *, char **);
for_rtl_init_ (&argc, argv);
```

FOR_SET_FPE

Run-Time Function: Sets the floating-point exception flags. This routine can be called from a C or Fortran program.

Module: USE DFLIB

Syntax

result = **FOR_SET_FPE** (*a*)

a
Must be of type INTEGER(4). It contains bit flags controlling floating-point exception trapping, reporting, and result handling.

Results:

The result type is INTEGER(4). The return value represents the previous settings of the floating-point exception flags. The meanings of the bits are defined in the DFLIB module file.

To get the current settings of the floating-point exception flags, use FOR_GET_FPE.

Example

Consider the following:

```
USE DFLIB

INTEGER*4 FOR_SET_FPE, OLD_FPE_FLAGS, NEW_FPE_FLAGS
EXTERNAL FOR_SET_FPE
OLD_FPE_FLAGS = FOR_SET_FPE (NEW_FPE_FLAGS)
```

FOR_SET_REENTRANCY

Run-Time Function: Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits. This routine can be called from a C or Fortran program.

Module: USE DFLIB

Syntax

result = **FOR_SET_REENTRANCY** (*mode*)

mode
Must be of type INTEGER(4) and contain one of the following options:

- FOR_K_REENTRANCY_NONE
  Tells the Fortran RTL to perform simple locking around critical sections of RTL code. This type of reentrancy should be used when the Fortran RTL will *not* be reentered due to asynchronous system traps (ASTs) or threads within the application.

- FOR_K_REENTRANCY_ASYNCH
  Tells the Fortran RTL to perform simple locking and disables ASTs around critical sections of RTL code. This type of reentrancy should be used when the application contains AST handlers that call the Fortran RTL.

- FOR_K_REENTRANCY_THREADED
  Tells the Fortran RTL to perform thread locking. This type of reentrancy should be used in multithreaded applications.

- FOR_K_REENTRANCY_INFO
  Tells the Fortran RTL to return the current reentrancy mode.

Results:

The result type is INTEGER(4). The return value represents the previous setting of the Fortran Run-Time Library reentrancy mode, unless the argument is FOR_K_REENTRANCY_INFO, in which case the return value represents the current setting.

You must be using an RTL that supports the level of reentrancy you desire. For example, **FOR_SET_REENTRANCY** ignores a request for thread protection (FOR_K_REENTRANCY_THREADED) if you do not build your program with the thread-safe RTL.

Example

Consider the following:

```
PROGRAM SETREENT
USE DFLIB

INTEGER*4    MODE
CHARACTER*10 REENT_TXT(3) /'NONE    ','ASYNCH  ','THREADED'/

PRINT*,'Setting Reentrancy mode to ',REENT_TXT(MODE+1)
MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_NONE)
PRINT*,'Previous Reentrancy mode was ',REENT_TXT(MODE+1)

MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_INFO)
PRINT*,'Current Reentrancy mode is ',REENT_TXT(MODE+1)

END
```

FORALL

Statement and Construct: The **FORALL** statement and construct is an element-by-element generalization of the Fortran 90 masked array assignment (WHERE statement and construct). It allows more general array shapes to be assigned, especially in construct form.

FORALL is a DIGITAL Fortran extension to Fortran 90, but it is a language feature of Fortran 95.

Syntax

Statement:
FORALL (*triplet-spec* [,*triplet-spec*]...[,*mask-expr*]) *assignment-stmt*

Construct:
[*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* is 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*.

assignment-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. The expression being assigned must not be a character expression.

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.
- 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

The following example, which is not expressible using array syntax, sets diagonal elements of an array to 1:

```
REAL, DIMENSION(N, N) :: A
FORALL (I=1:N)  A(I, I) = 1
```

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.

FORMAT

Statement: Specifies the form of data being transferred and the data conversion (editing) required to achieve that form.

Syntax

FORMAT (*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, :, /, $, \, 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 the OPEN statement.

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, $, +, 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.

Whenever an edit descriptor requires an integer constant, you can specify an integer expression in a **FORMAT** statement. The integer expression must be enclosed by angle brackets (< and >). The following examples are valid format specifications:

```
    WRITE(6,20) INT1
20  FORMAT(I<MAX(20,5)>)

    WRITE(6,FMT=30) INT2, INT3
30  FORMAT(I<J+K>, I<2*M>)
```

The integer expression can be any valid Fortran expression, including function calls and references to dummy arguments, with the following restrictions:

- Expressions cannot be used with the **H** edit descriptor.
- Expressions cannot contain graphical relational operators (such as > and <).

The value of the expression is reevaluated each time an input/output item is processed during the execution of the **READ**, **WRITE**, or **PRINT** statement.

The following table summarizes the edit descriptors:

Data Edit Descriptors		
Code	**Form [1]**	**Effect**
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
D	Dw.d	Transfers real values with D exponents.
E	Ew.d[Ee]	Transfers real values with E exponents.
EN	ENw.d [Ee]	Transfers real values with engineering notation.
ES	ESw.d [Ee]	Transfers real values with scientific notation.
F	Fw.d	Transfers real values with no exponent.
G	Gw.d[Ee]	Transfers values of all intrinsic types.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
Z	Zw[.m]	Transfers hexadecimal values.

```
[1] w is the field width
    m is the minimum number of digits that must be in the field (including zeros).
    d is the number of digits to the right of the decimal point
    E is the exponent field
    e is the number of digits in the exponent
```

Control Edit Descriptors		
Code	**Form**	**Effect**
BN	BN	Ignores embedded and trailing blanks in a numeric input field.

BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.
TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
$	$	Suppresses trailing carriage return during interactive I/O.
:	:	Terminates format control if there are no more items in the I/O list.
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.
String Edit Descriptors		
Code	**Form**	**Effect**
H	nHch [ch...]	Transfers characters following the H edit descriptor to an output record.
'c' [2]	'c'	Transfers the character literal constant (between the delimiters) to an output record.
[2] These delimiters can also be quotation marks (").		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: I/O Formatting, Format Specifications, Data Edit Descriptors

Example

```
   INTEGER width, value
   width = 2
   read (*,1) width, value
!  if the input is 3123, prints 123, not 12
1  format ( i1, i<width>)
```

```
print *, value
END
```

FP_CLASS

Elemental Intrinsic Function (Generic): Returns the class of an IEEE® real (S_floating, T_floating, or X_floating) argument.

Syntax

result = **FP_CLASS** (*x*)

x
(Input) Must be of type real.

Results:

The result type is 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 return values are defined in file fordef.for in \DF98\INCLUDE.

Example

FP_CLASS (4.0_8) has the value 4 (FOR_K_FP_POS_NORM).

FPUTC

Portability Function: Writes a character to the file specified by a Fortran external unit, bypassing normal Fortran input/output.

Module: USE DFPORT

Syntax

result = **FPUTC** (*lunit*, *char*)

lunit
(Input) INTEGER(4). Unit number of a file.

char
(Output) Character*(*). Variable whose value is to be written to the file corresponding to *lunit*.

Results:

The result type is INTEGER(4). The result is zero if the write was successful; otherwise, an error code, such as:

EINVAL The specified unit is invalid (either not already open, or an invalid unit number)

If you use **WRITE**, **READ**, or any other Fortran I/O statements with *lunit*, be sure to read Input and Output With Portability Routines in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: I/O Formatting, Files, Devices, and Input/Output Hardware

Example

```
use dfport
integer*4 lunit, i4
character*26 string
character*1 char1
lunit = 1
open (lunit,file = 'fputc.dat')
do i = 1,26
  char1 = char(123-i)
  i4 = fputc(1,char1)        !make valid writes
  if (i4.ne.0) iflag = 1
enddo
rewind (1)
read (1,'(a)') string
print *, string
```

FRACTION

Elemental Intrinsic Function (Generic): Returns the fractional part of the model representation of the

argument value.

Syntax

result = **FRACTION** (*x*)

x
(Input) Must be of type real.

Results:

The result type is the same as *x*. The result has the value *x* x b^e. Parameters *b* and *e* are defined in Model for Real Data. If *x* has the value zero, the result has the value zero.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DIGITS, RADIX, EXPONENT, Data Representation Models

Examples

If 3.0 is a REAL(4) value, FRACTION (3.0) has the value 0.75.

The following shows another example:

```
REAL result
result = FRACTION(3.0) ! returns 0.75
result = FRACTION(1024.0) ! returns 0.5
```

FREE

Intrinsic Subroutine: Frees a block of memory that is currently allocated.

Syntax

CALL FREE (*i*)

i
(Input) Must be of type INTEGER(4) on Intel processors; INTEGER(8) on Alpha processors. This value is the starting address of the memory block to be freed, previously allocated by MALLOC.

If the freed address was not previously allocated by **MALLOC**, or if an address is freed more than once, results are unpredictable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
INTEGER(4) addr, size
size = 1024             ! size in bytes
addr = MALLOC(size)     ! allocate the memory
CALL FREE(addr)         ! free it
END
```

FREEFORM and NOFREEFORM

Compiler Directives: FREEFORM specifies that source code is in free-form format. **NOFREEFORM** specifies that source code is in fixed-form format.

Syntax

*c***DEC$ FREEFORM**
*c***DEC$ NOFREEFORM**

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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.

The following forms are also allowed: `!MS$FREEFORM` and `!MS$NOFREEFORM`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Source Forms, General Compiler Directives, /free

FSEEK

Portability Function: Repositions a file specified by a Fortran external unit.

Module: USE DFPORT

Syntax

result = **FSEEK** (*lunit*, *offset*, *from*)

lunit
(Input) INTEGER(4). External unit number of a file.

offset
(Input) INTEGER(4). Offset in bytes, relative to *from*, that is to be the new location of the file

marker.

from
(Input) INTEGER(4). A position in the file. Portability defines the following parameters:

- SEEK_SET = 0 - Beginning of the file
- SEEK_CUR = 1 - Current position
- SEEK_END = 2 - End of the file

Results:

The result type is INTEGER(4). The result is zero if the repositioning was successful; otherwise, an error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number), or the *from* parameter is invalid.

The file specified in *lunit* must be open.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFPORT
integer(4) istat, offset, ipos
character ichar
OPEN (unit=1,file='datfile.dat')
offset = 5
ipos = 0
istat=fseek(1,offset,ipos)
if (.NOT. stat) then
  istat=fgetc(1,ichar)
  print *, 'data is ',ichar
end if
```

FSTAT

Portability Function: Returns detailed information about a file specified by a external unit number.

Module: USE DFPORT

Syntax

result = **FSTAT** (*lunit*, *statb*)

lunit
(Input) INTEGER(4). External unit number of the file to examine.

statb

(Output) INTEGER(4). One-dimensional array with a size of 12. The following table describes the elements of the array:

statb **element**	**Return value**
statb(1)	Device the file resides on (always 0)
statb(2)	Inode number (always 0)
statb(3)	File type, attributes, and access control information (see the following table)
statb(4)	Number of links (always 1)
statb(5)	User ID of owner (always 1)
statb(6)	Group ID of owner (always 1)
statb(7)	Raw device file resides on (always 1)
statb(8)	The size of the file in bytes
statb(9)	The time of last access (only available on non-FAT file systems; same as statb(10) on FAT systems.
statb(10)	The time of last modification
statb(11)	The time of last status change (same as statb(10))
statb(12)	Block size (always 1)

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, returns an error code equal to EINVAL (*lunit* is not a valid unit number, or is not open).

Mode is a bitmap consisting of an **IOR** of the following constants (the module DFPORT supplies parameters with the symbolic names given):

Symbolic name	**Constant**	**Description**	**Notes**
S_IFMT	O'0170000'	Type of file	
S_IFDIR	O'0040000'	Directory	
S_IFCHR	O'0020000'	Character special	Never set
S_IFBLK	O'0060000'	Block special	Never set
S_IFREG	O'0100000'	Regular	

S_IFLNK	O’0120000’	Symbolic link	Never set
S_IFSOCK	O’0140000’	Socket	Never set
S_ISUID	O’0004000’	Set user ID on execution	Never set
S_ISGID	O’0002000’	Set group ID on execution	Never set
S_ISVTX	O’0001000’	Save swapped text	Never set
S_IRWXU	O’0000700’	Owner’s file permissions	
S_IRUSR, S_IREAD	O’0000400’	Owner read permission	Always true
S_IWUSR, S_IWRITE	O’0000200’	Owner write permission	
S_IXUSR, S_IEXEC	O’0000100’	Owner execute permission	Set if S_IREAD is set
S_IRWXG	O’0000070’	Group’s file permissions	Same as S_IRWXU
S_IRGRP	O’0000040’	Group read permission	Same as S_IRUSR
S_IWGRP	O’0000020’	Group write permission	Same as S_IWUSR
S_IXGRP	O’0000010’	Group execute permission	Same as S_IXUSR
S_IRWXO	O’0000007’	Other’s file permissions	Same as S_IRWXU
S_IROTH	O’0000004’	Other’s read permission	Same as S_IRUSR
S_IWOTH	O’0000002’	Other write permission	Same as S_IWUSR
S_IXOTH	O’0000001’	Other execute permission	Same as S_IXUSR

Time values are returned as number of seconds since 0:00:00 Greenwich mean time, January 1, 1970.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INQUIRE

Example

```
USE DFPORT
integer(4) statarray(12), istat
OPEN (unit=1,file=’datfile.dat’)
ISTAT = FSTAT (1, statarray)
if (.NOT. istat) then
    print *, statarray
```

```
end if
```

FTELL

Portability Function: Returns the current position of a file.

Module: USE DFPORT

Syntax

result = **FTELL** (*lunit*)

lunit
(Input) INTEGER(4). External unit number of a file.

Results:

The result type is INTEGER(4). The result is the offset, in bytes, from the beginning of the file. A negative value indicates an error, which is the negation of the **IERRNO** error code. The following is an example of an error code:

EINVAL: *lunit* is not a valid unit number, or is not open.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
C  An END statement must be the last statement in a program
C  unit:
   PROGRAM MyProg
   WRITE (*, '("Hello, world!")')
   END
C
C  An example of a named subroutine
C
   SUBROUTINE EXT1 (X,Y,Z)
   Real, Dimension (100,100) :: X, Y, Z
   END SUBROUTINE EXT1
```

FULLPATHQQ

Run-Time Function: Returns the full path for a specified file or directory.

Module: USE DFLIB

Syntax

result = **FULLPATHQQ** (*name*, *pathbuf*)

name
(Input) Character*(*). Item for which you want the full path. Can be the name of a file in the current directory, a relative directory or filename, or a network uniform naming convention (UNC) path.

pathbuf
(Output) Character*(*). Buffer to receive full path of the item specified in *name*.

Results:

The result type is INTEGER(4). The result is the length of the full path in bytes, or 0 if the function fails (usually for an invalid name).

The length of the full path depends upon how deeply the directories are nested on the drive you are using. If the full path is longer than the character buffer provided to return it (*pathbuf*), **FULLPATHQQ** returns only that portion of the path that fits into the buffer.

Check the length of the path before using the string returned in *pathbuf*. If the longest full path you are likely to encounter does not fit into the buffer you are using, allocate a larger character buffer. You can allocate the largest possible path buffer with the following statements:

```
USE DFLIB
CHARACTER($MAXPATH) pathbuf
```

$MAXPATH is a symbolic constant defined in module DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as 260.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SPLITPATHQQ

Example

```
USE DFLIB
CHARACTER($MAXPATH) buf
CHARACTER(3)        drive
CHARACTER(256)      dir
CHARACTER(256)      name
CHARACTER(256)      ext
CHARACTER(256)      file

INTEGER(4)          len

DO WHILE (.TRUE.)
  WRITE (*,*)
  WRITE (*,'(A, \)') ' Enter filename (Hit &
                      RETURN to exit): '
  len = GETSTRQQ(file)
  IF (len .EQ. 0) EXIT
  len = FULLPATHQQ(file, buf)
  IF (len .GT. 0) THEN
```

```
          WRITE (*,*) buf(:len)
        ELSE
          WRITE (*,*) 'Can''t get full path'
          EXIT
        END IF
!
!       Split path
        WRITE (*,*)
        len = SPLITPATHQQ(buf, drive, dir, name, ext)
        IF (len .NE. 0) THEN
          WRITE (*, 900) ' Drive: ', drive
          WRITE (*, 900) ' Directory: ', dir(1:len)
          WRITE (*, 900) ' Name: ', name
          WRITE (*, 900) ' Extension: ', ext
        ELSE
          WRITE (*, *) 'Can''t split path'
        END IF
      END DO
900   FORMAT (A, A)
      END
```

FUNCTION

Statement: The initial statement of a function subprogram. A function subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression.

Syntax

[*prefix*] **FUNCTION** *name* ([*d-arg-list*]) [**RESULT** (*r-name*)]

prefix
(Optional) Is one of the following:

type [*keyword*]
keyword [*type*]

type
Is a data type specifier.

keyword
Is **RECURSIVE**, **PURE**, or **ELEMENTAL**.

The keyword **RECURSIVE** indicates a recursive function. A recursive function is one that calls itself or calls another subprogram, which in turn calls the first function before the first function has completed execution. The keyword must appear in the function declaration if the function calls itself either directly or indirectly.

The keyword PURE asserts that the procedure has no side effects. The keyword ELEMENTAL indicates a restricted form of pure procedure.

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
(Optional) Is a list of one or more dummy arguments.

r-name
(Optional) 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. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ENTRY, SUBROUTINE, PURE, ELEMENTAL, RESULT Keyword, Function References, Program Units and Procedures, General Rules for Function and Subroutine Subprograms

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
```

The following shows another example:

```
      INTEGER Divby2
10   PRINT *, 'Enter a number'
      READ *, i
      Print *, Divby2(i)
      GOTO 10
      END
C
C     This is the function definition
C
      INTEGER FUNCTION Divby2 (num)
      Divby2=num / 2
      END FUNCTION
```

GERROR

Portability Subroutine: Returns a message for the last error detected by a Fortran run-time routine.

Syntax

CALL GERROR (*string*)

string
(Output) Character*(*). Message corresponding to the last detected error.

The last detected error does not necessarily correspond to the most recent function call. Visual Fortran resets *string* only when another error occurs.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PERROR, IERRNO

Example

```
USE DFPORT
character*40 errtext
character char1
integer*4 iflag, i4
. . .!Open unit 1 here
i4=fgetc(1,char1)
if (i4) then
  iflag = 1
  Call GERROR (errtext)
  print *, errtext
end if
```

GETACTIVEQQ

QuickWin Function: Returns the unit number of the currently active child window.

Module: USE DFLIB

Syntax

result = **GETACTIVEQQ** ( )

Results:

The result type is INTEGER(4). The result is the unit number of the currently active window. Returns the parameter QWIN$NOACTIVEWINDOW (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) if no child window is active.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: SETACTIVEQQ, GETHWNDQQ, Using QuickWin.

GETARCINFO

Graphics Function: Determines the endpoints (in viewport coordinates) of the most recently drawn arc or pie.

Module: USE DFLIB

Syntax

result = **GETARCINFO** (*pstart*, *pend*, *ppaint*)

pstart
(Output) Derived type xycoord. Viewport coordinates of the starting point of the arc.

pend
(Output) Derived type xycoord. Viewport coordinates of the end point of the arc.

ppaint
(Output) Derived type xycoord. Viewport coordinates of the point at which the fill begins.

Results:

The result type is INTEGER(2). The result is nonzero if successful. The result is zero if neither the **ARC** nor the **PIE** function has been successfully called since the last time **CLEARSCREEN** or **SETWINDOWCONFIG** was successfully called, or since a new viewport was selected.

GETARCINFO updates the *pstart* and *pend* **xycoord** derived types to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to the **ARC** or **PIE** functions. The xycoord derived type, defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory), is:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

The returned value in *ppaint* specifies a point from which a pie can be filled. You can use this to fill a pie in a color different from the border color. After a call to **GETARCINFO**, change colors using **SETCOLORRGB**. Use the new color, along with the coordinates in *ppaint*, as arguments for the **FLOODFILLRGB** function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: ARC, FLOODFILLRGB, GETCOLORRGB, GRSTATUS, PIE, SETCOLORRGB

Example

```
USE DFLIB
INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4
TYPE (xycoord) xystart, xyend, xyfillpt
x1 = 80; y1 = 50
x2 = 240; y2 = 150
x3 = 120; y3 = 80
x4 = 90; y4 = 180

status = ARC(x1, y1, x2, y2, x3, y3, x4, y4)
status = GETARCINFO(xystart, xyend, xyfillpt)
END
```

GETARG

Run-Time Subroutine: Returns the specified command-line argument (where the command itself is argument number 0).

Module: USE DFLIB

Syntax

CALL GETARG (*n*, *buffer* [, *status*])

n
(Input) INTEGER(2). Position of the command-line argument to retrieve. The command itself is argument number 0.

buffer
(Output) Character*(*). Command-line argument retrieved.

status
(Optional; output) INTEGER(2). If specified, returns the completion status. If there were no errors, *status* returns the number of characters in the retrieved command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors return a value of -1. Errors include specifying an argument position less than 0 or greater than the value returned by **NARGS**.

GETARG can be used with two or three arguments. If you use module DFLIB.F90 in the \DF98 \INCLUDE subdirectory (by including the statement **USE DFLIB**), you can mix calls to **GETARG** with two or three arguments. If you do not use DFLIB.F90, you can use either two-argument or three-argument calls to **GETARG** but only one type of call within a subprogram.

GETARG returns command-line arguments as they were entered. There is no case conversion.

If the command-line argument is shorter than *buffer*, **GETARG** pads *buffer* on the right with blanks. If the argument is longer than *buffer*, **GETARG** truncates the argument. If there is an error, **GETARG** fills *buffer* with blanks.

Assume a command-line invocation of ANOVA -g -c -a, and that *buffer* is at least five characters long. The following **GETARG** statements return the corresponding arguments in *buffer*:

Statement	**String returned in *buffer***	**Length returned in *status***
CALL GETARG (0, buffer, status)	ANOVA	5
CALL GETARG (1, buffer)	-g	undefined
CALL GETARG (2, buffer, status)	-c	2
CALL GETARG (3, buffer)	-a	undefined
CALL GETARG (4, buffer, status)	all blanks	-1

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NARGS, IARGC

Example

```
USE DFLIB
INTEGER(2) result
result = RUNQQ('prog', '-c -r')
END
! PROG.F90
USE DFLIB
INTEGER(2) n1, n2, status
CHARACTER(80) buf
n1 = 1
n2 = 2
CALL GETARG(n1, buf, status)
WRITE(*,*) buf
CALL GETARG(n2, buf )
WRITE (*,*) buf
END
```

GETBKCOLOR

Graphics Function: Gets the current background color index for both text and graphics output.

Module: USE DFLIB

Syntax

result = **GETBKCOLOR** ( )

Results:

The result type is INTEGER(4). The result is the current background color index.

GETBKCOLOR returns the current background color index for both text and graphics, as set with **SETBKCOLOR**. The color index of text over the background color is set with **SETTEXTCOLOR** and returned with **GETTEXTCOLOR**. The color index of graphics over the background color is set with **SETCOLOR** and returned with **GETCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use **SETBKCOLORRGB**, **SETCOLORRGB**, and **SETTEXTCOLORRGB**.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are **GETBKCOLOR** and **SETBKCOLOR**. The default background index is 0, which is associated with black unless the user remaps the palette with **REMAPPALETTERGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETBKCOLORRGB, SETBKCOLOR, GETCOLOR, GETTEXTCOLOR, REMAPALLPALETTERGB, REMAPPALETTERGB

Example

```
USE DFLIB
INTEGER(4) bcindex
bcindex = GETBKCOLOR()
```

GETBKCOLORRGB

Graphics Function: Gets the current background Red-Green-Blue (RGB) color value for both text and graphics.

Module: USE DFLIB

Syntax

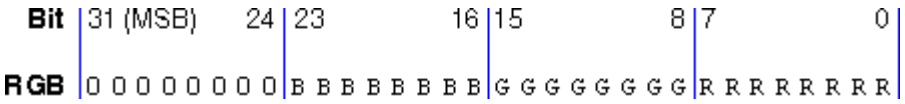
result = **GETBKCOLORRGB** ( )

Results:

The result type is INTEGER(4). The result is the RGB value of the current background color for both text and graphics.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit

value (2 hex digits). In the value you retrieve with **GETBKCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with **SETBKCOLORRGB**. The RGB color value of text over the background color (used by text functions such as **OUTTEXT**, **WRITE**, and **PRINT**) is set with **SETTEXTCOLORRGB** and returned with **GETTEXTCOLORRGB**. The RGB color value of graphics over the background color (used by graphics functions such as **ARC**, **OUTGTEXT**, and **FLOODFILLRGB**) is set with **SETCOLORRGB** and returned with **GETCOLORRGB**.

SETBKCOLORRGB (and the other RGB color selection functions **SETCOLORRGB** and **SETTEXTCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (**SETBKCOLOR**, **SETCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCOLORRGB, GETTEXTCOLORRGB, SETBKCOLORRGB, GETBKCOLOR

Example

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(4) back, fore, oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
oldcolor = SETCOLORRGB(#FF) ! red
! reverse the screen
back = GETBKCOLORRGB()
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)
oldcolor = SETCOLORRGB(back)
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

GETC

Portability Function: Reads the next available character from external unit 5, which is normally connected to the console.

Module: USE DFPORT

Syntax

result = **GETC** (*char*)

char
(Output) Character*(*). First character typed at the keyboard after the call to **GETC**. If unit 5 is connected to a console device, then no characters are returned until the Enter key is pressed.

Results:

The result type is INTEGER(4). The result is zero if successful, or -1 if an end-of-file was detected.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAHICS WINDOWS DLL LIB

See Also: GETCHARQQ, GETSTRQQ, *Programmer's Guide*: Portability Library

Example

```
     use dfport
     character ans,errtxt*40
     print *, 'Enter a character: '
     ISTAT = GETC (ans)

if (istat) then
       call gerror(errtxt)
     end if
```

GETCHARQQ

Run-Time Function: Gets the next keystroke.

Module: USE DFLIB

Syntax

result = **GETCHARQQ** ( )

Results:

The result type is CHARACTER(1). The result is the character representing the key that was pressed. The value can be any ASCII character.

If the key pressed is a represented by a single ASCII character, **GETCHARQQ** returns the character. If the key pressed is a function or direction key, a hex #00 or #E0 is returned. If you need to know which function or direction was pressed, call **GETCHARQQ** a second time to get the extended code for the key.

If there is no keystroke waiting in the keyboard buffer, **GETCHARQQ** waits until there is one, and then returns it. Compare this to the function **PEEKCHARQQ**, which returns .TRUE. if there is a character waiting in the keyboard buffer and .FALSE. if not. You can use **PEEKCHARQQ** to determine if **GETCHARQQ** should be called. This can prevent a program from hanging while **GETCHARQQ** waits for a keystroke that isn't there.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PEEKCHARQQ, GETSTRQQ, INCHARQQ, MBINCHARQQ, GETC, FGETC

Example

```
!  Program to demonstrate GETCHARQQ
USE DFLIB
CHARACTER(1) key / 'A' /
PARAMETER (ESC = 27)
PARAMETER (NOREP = 0)
WRITE (*,*) ' Type a key: (or q to quit)'
!  Read keys until ESC or q is pressed
DO WHILE (ICHAR (key) .NE. ESC)
   key = GETCHARQQ()
!  Some extended keys have no ASCII representation
   IF(ICHAR(key) .EQ. NOREP) THEN
     key = GETCHARQQ()
     WRITE (*, 900) 'Not ASCII. Char = NA'
     WRITE (*,*)
!  Otherwise, there is only one key
   ELSE
     WRITE (*,900) 'ASCII. Char = '
     WRITE (*,901) key
   END IF
   IF (key .EQ. 'q' ) THEN
     EXIT
   END IF
   END DO
900   FORMAT (1X, A, \)
901   FORMAT (A)
END
```

GETCOLOR

Graphics Function: Gets the current graphics color index.

Module: USE DFLIB

Syntax

result = **GETCOLOR ( )**

Results:

The result type is INTEGER(2). The result is the current color index, if successful; otherwise, - 1.

GETCOLOR returns the current color index used for graphics over the background color as set with **SETCOLOR**. The background color index is set with **SETBKCOLOR** and returned with **GETBKCOLOR**. The color index of text over the background color is set with **SETTEXTCOLOR** and returned with **GETTEXTCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use **SETCOLORRGB**, **SETBKCOLORRGB**, and **SETTEXTCOLORRGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCOLORRGB, GETBKCOLOR, GETTEXTCOLOR, SETCOLOR

Example

```
!  Program to demonstrate GETCOLOR
 PROGRAM COLORS
 USE DFLIB
 INTEGER(2) loop, loop1, status, color
 LOGICAL(4) winstat
 REAL rnd1, rnd2, xnum, ynum
 type (windowconfig) wc
 status = SETCOLOR(INT2(0))
!  Color random pixels with 15 different colors
 DO loop1 = 1, 15
   color = INT2(MOD(GETCOLOR() +1, 16))
   status = SETCOLOR (color) ! Set to next color
   DO loop = 1, 75
!  Set color of random spot, normalized to be on screen
     CALL RANDOM(rnd1)
     CALL RANDOM(rnd2)
     winstat = GETWINDOWCONFIG(wc)
     xnum = wc.numxpixels
     ynum = wc.numypixels
     status = &
     SETPIXEL(INT2(rnd1*xnum+1),INT2(rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2(rnd2*ynum+1))
     status = &
     SETPIXEL(INT2(rnd1*xnum-1),INT2( rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2( rnd2*ynum-1))
   END DO
 END DO
 END
```

GETCOLORRGB

Graphics Function: Gets the current graphics color Red-Green-Blue (RGB) value (used by graphics functions such as **ARC, ELLIPSE**, and **FLOODFILLRGB**).

Module: USE DFLIB

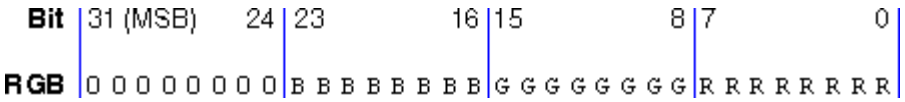
Syntax

result = **GETCOLORRGB** ( )

Results:

The result type is INTEGER(4). The result is the RGB value of the current graphics color.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with **GETCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETCOLORRGB returns the RGB color value of graphics over the background color (used by graphics functions such as **ARC, ELLIPSE**, and **FLOODFILLRGB**), set with **SETCOLORRGB**. **GETBKCOLORRGB** returns the RGB color value of the current background for both text and graphics, set with **SETBKCOLORRGB**. **GETTEXTCOLORRGB** returns the RGB color value of text over the background color (used by text functions such as **OUTTEXT, WRITE**, and **PRINT**), set with **SETTEXTCOLORRGB**.

SETCOLORRGB (and the other RGB color selection functions **SETBKCOLORRGB** and **SETTEXTCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (**SETCOLOR**, **SETBKCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETBKCOLORRGB, GETTEXTCOLORRGB, SETCOLORRGB, GETCOLOR

Example

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(2) numfonts
INTEGER(4) fore, oldcolor

numfonts = INITIALIZEFONTS ( )
oldcolor = SETCOLORRGB(#FF)   ! set graphics
                              ! color to red
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)  ! set background
                                ! to graphics color
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB (#FF0000) ! set graphics
                                 ! color to blue

CALL OUTGTEXT("hello, world")
END
```

GETCONTROLFPQQ (x86 only)

Run-Time Subroutine: Returns the floating-point processor control word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL GETCONTROLFPQQ (*controlword*)

controlword
(Output) INTEGER(2). Floating-point processor control word.

The floating-point control word is a bit flag that controls various modes of the floating-point coprocessor. The DFLIB.F90 module file (in the \DF98\INCLUDE subdirectory) contains constants defined for the control word as follows:

Parameter name	Hex value	Description
FPCW$MCW_IC	#1000	**Infinity control mask**
FPCW$AFFINE	#1000	Affine infinity
FPCW$PROJECTIVE	#0000	Projective infinity
FPCW$MCW_PC	#0300	**Precision control mask**
FPCW$64	#0300	64-bit precision
FPCW$53	#0200	53-bit precision
FPCW$24	#0000	24-bit precision

FPCW$MCW_RC	#0C00	**Rounding control mask**
FPCW$CHOP	#0C00	Truncate
FPCW$UP	#0800	Round up
FPCW$DOWN	#0400	Round down
FPCW$NEAR	#0000	Round to nearest
FPCW$MSW_EM	#003F	**Exception mask**
FPCW$INVALID	#0001	Allow invalid numbers
FPCW$DENORMAL	#0002	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	#0004	Allow divide by zero
FPCW$OVERFLOW	#0008	Allow overflow
FPCW$UNDERFLOW	#0010	Allow underflow
FPCW$INEXACT	#0020	Allow inexact precision

The defaults for the floating-point control word are 53-bit precision, round to nearest, and the denormal, underflow and inexact precision exceptions disabled. An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0. Exceptions can be disabled by setting the flags to 1 with **SETCONTROLFPQQ**.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues.

You can find out which exceptions (if any) occurred by calling **GETSTATUSFPQQ**. If errors on floating-point exceptions are enabled (by clearing the flags to 0 with **SETCONTROLFPQQ**), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with **SIGNALQQ** and branch to your own error-handling routines.

You can use **GETCONTROLFPQQ** to retrieve the current control word and **SETCONTROLFPQQ** to change the control word. Most users do not need to change the default settings. For a full discussion of the floating-point control word, exceptions, and error handling, see The Floating-Point Environment in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SETCONTROLFPQQ, GETSTATUSFPQQ, SIGNALQQ, MATHERRQQ

Example

```
USE DFLIB
INTEGER(2) control
CALL GETCONTROLFPQQ (control)
     !if not rounding down
IF (IAND(control, FPCW$DOWN) .NE. FPCW$DOWN) THEN
  control = IAND(control, NOT(FPCW$MCW_RC)) ! clear all
                                            !  rounding
  control = IOR(control, FPCW$DOWN)         !  set to
                                            !  round down
  CALL SETCONTROLFPQQ(control)
END IF
END
```

GETCWD

Portability Function: Retrieves the path of the current working directory.

Module: USE DFPORT

Syntax

result = **GETCWD** (*dirname*)

dirname
(Output) Character *(*). Name of the current working directory path, including drive letter.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDRIVEDIRQQ

Example

```
    character*30 dirname
!   variable dirname must be long enough to hold entire string
    integer(4) istat
    ISTAT = GETCWD (dirname)
    IF (ISTAT == 0) write *, 'Current directory is ',dirname
```

GETCURRENTPOSITION, GETCURRENTPOSITION_W

Graphics Subroutines: Get the coordinates of the current graphics position.

Module: USE DFLIB

Syntax

CALL GETCURRENTPOSITION (*t*)
CALL GETCURRENTPOSITION_W (*wt*)

t
(Output) Derived type xycoord. Viewport coordinates of current graphics position. The derived type xycoord is defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

wt
(Output) Derived type wxycoord. Window coordinates of current graphics position. The derived type wxycoord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE wxycoord
  REAL(8) wx    ! x-coordinate
  REAL(8) wy    ! y-coordinate
END TYPE wxycoord
```

LINETO, MOVETO, and **OUTGTEXT** all change the current graphics position. It is in the center of the screen when a window is created.

Graphics output starts at the current graphics position returned by **GETCURRENTPOSITION** or **GETCURRENTPOSITION_W.** This position is not related to normal text output (from **OUTTEXT** or **WRITE**, for example), which begins at the current text position (see **SETTEXTPOSITION**). It does, however, affect graphics text output from **OUTGTEXT**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: LINETO, MOVETO, OUTGTEXT, SETTEXTPOSITION, GETTEXTPOSITION

Example

```
! Program to demonstrate GETCURRENTPOSITION
USE DFLIB
TYPE (xycoord) position
INTEGER(2) result
result = LINETO(INT2(300), INT2(200))
CALL GETCURRENTPOSITION( position )
IF (position.xcoord .GT. 50) THEN
  CALL MOVETO(INT2(50), position.ycoord, position)
  WRITE(*,*) "Text unaffected by graphics position"
END IF
result = LINETO(INT2(300), INT2(200))
END
```

GETDAT

Run-Time Subroutine: Returns the date.

Module: USE DFLIB

Syntax

CALL GETDAT (*iyr*, *imon*, *iday*)

iyr
(Output) INTEGER(2). Year (*xxxx* AD).

imon
(Output) INTEGER(2). Month (1-12).

iday
(Output) INTEGER(2). Day of the month (1-31).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: GETTIM, SETDAT, SETTIM, DATE, FDATE, IDATE, JDATE

Example

```
! Program to demonstrate GETDAT and GETTIM
USE DFLIB
INTEGER(2) tmpday, tmpmonth, tmpyear
INTEGER(2) tmphour, tmpminute, tmpsecond, tmphund
CHARACTER(1) mer

CALL GETDAT(tmpyear, tmpmonth, tmpday)
CALL GETTIM(tmphour, tmpminute, tmpsecond, tmphund)
IF (tmphour .GT. 12) THEN
  mer = 'p'
  tmphour = tmphour - 12
ELSE
  mer = 'a'
END IF
WRITE (*, 900) tmpmonth, tmpday, tmpyear
900  FORMAT(I2, '/', I2.2, '/', I4.4)
WRITE (*, 901) tmphour,tmpminute,tmpsecond,tmphund,mer
901  FORMAT(I2, ':', I2.2, ':', I2.2, ':', I2.2, ' ',&
            A, 'm')
END
```

GETDRIVEDIRQQ

Run-Time Function: Gets the path of the current working directory on a specified drive.

Module: USE DFLIB

Syntax

result = **GETDRIVEDIRQQ** (*drivedir*)

drivedir
(Input; output) Character*(*). On input, drive whose current working directory path is to be returned. On output, string containing the current directory on that drive in the form d:\dir.

Results:

The result type is INTEGER(4). The result is the length (in bytes) of the full path of the directory on the specified drive. Zero is returned if the path is longer than the size of the character buffer *drivedir*.

You specify the drive from which to return the current working directory by putting the drive letter into *drivedir* before calling **GETDRIVEDIRQQ**. To make sure you get information about the current drive, put the symbolic constant **FILE$CURDRIVE** (defined in DFLIB.F90 in the \DF98 \INCLUDE subdirectory) into *drivedir*.

Because drives are identified by a single alphabetic character, **GETDRIVEDIRQQ** examines only the first letter of *drivedir*. For instance, if *drivedir* contains the path `c:\fps90\bin`, **GETDRIVEDIRQQ** (*drivedir*) returns the current working directory on drive C and disregards the rest of the path. The drive letter can be uppercase or lowercase.

The length of the path returned depends on how deeply the directories are nested on the drive specified in *drivedir*. If the full path is longer than the length of *drivedir*, **GETDRIVEDIRQQ** returns only the portion of the path that fits into *drivedir*. If you are likely to encounter a long path, allocate a buffer of size $MAXPATH ($MAXPATH = 260).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: CHANGEDRIVEQQ, CHANGEDIRQQ, GETDRIVESIZEQQ, GETDRIVESQQ, GETLASTERRORQQ, SPLITPATHQQ

Example

```
!  Program to demonstrate GETDRIVEDIRQQ
USE DFLIB
CHARACTER($MAXPATH) dir
INTEGER(4) length

!  Get current directory
dir = FILE$CURDRIVE
length = GETDRIVEDIRQQ(dir)
IF (length .GT. 0) THEN
  WRITE (*,*) 'Current directory is: '
  WRITE (*,*) dir
ELSE
  WRITE (*,*) 'Failed to get current directory'
END IF
```

```
END
```

GETDRIVESIZEQQ

Run-Time Function: Gets the total size of the specified drive and space available on it.

Module: USE DFLIB

Syntax

result = **GETDRIVESIZEQQ** (*drive*, *total*, *avail*)

drive
(Input) Character*(*). String containing the letter of the drive to get information about.

total
(Output) INTEGER(4). Total number of bytes on the drive.

avail
(Output) INTEGER(4). Number of bytes of available space on the drive.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Because drives are identified by a single alphabetic character, **GETDRIVESIZEQQ** examines only the first letter of *drive*. The drive letter can be uppercase or lowercase. You can use the constant FILE$CURDRIVE (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) to get the size of the current drive.

If **GETDRIVESIZEQQ** fails, use **GETLASTERRORQQ** to determine the reason.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETLASTERRORQQ, GETDRIVESQQ, GETDRIVEDIRQQ, CHANGEDRIVEQQ, CHANGEDIRQQ

Example

```
! Program to demonstrate GETDRIVESQQ and GETDRIVESIZEQQ
USE DFLIB
CHARACTER(26) drives
CHARACTER(1) adrive
LOGICAL(4) status
INTEGER(4) total, avail
INTEGER(2) i
! Get the list of drives
drives = GETDRIVESQQ()
WRITE (*,'(A, A)') ' Drives available: ', drives
```

```
!
!Cycle through them for free space and write to console
DO i = 1, 26
  adrive = drives(i:i)
  status = .FALSE.
  WRITE (*,'(A, A, A, \)') ' Drive ', CHAR(i + 64), ':'
  IF (adrive .NE. ' ') THEN
     status = GETDRIVESIZEQQ(adrive, total, avail)
  END IF
  IF (status) THEN
     WRITE (*,*) avail, ' of ', total, ' bytes free.'
  ELSE
     WRITE (*,*) 'Not available'
  END IF
END DO
END
```

GETDRIVESQQ

Run-Time Function: Reports which drives are available to the system.

Module: USE DFLIB

Syntax

result = **GETDRIVESQQ** ( )

Results:

The result is CHARACTER(26). It is the positional character string containing the letters of the drives available in the system.

The returned string contains letters for drives that are available, and blanks for drives that are not available. For example, on a system with A, C, and D drives, the string ACD' is returned.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: GETDRIVEDIRQQ, GETDRIVESIZEQQ, CHANGEDRIVEQQ

Example

See the example for GETDRIVESIZEQQ.

GETENV

Portability Subroutine: Retrieves the value of an environment variable.

Module: USE DFPORT

Syntax

CALL GETENV (*ename*, *evalue*)

ename
(Input) Character*(*). Environment variable to search for.

evalue
(Output) Character*(*). Value found for *ename*. Blank if *ename* is not found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETENVQQ, SETENVQQ

Example

```
use dfport
character*40 libname
CALL GETENV ("LIB",libname)
TYPE *, "The LIB variable points to ",libname
```

GETENVQQ

Run-Time Function: Gets the value of a specified environment variable from the current environment.

Module: USE DFLIB

Syntax

result = **GETENVQQ** (*varname*, *value*)

varname
(Input) Character*(*). Name of environment variable.

value
(Output) Character*(*). Value of the specified environment variable, in uppercase.

Results:

The result type is INTEGER(4). The result is the length of the string returned in *value*. Zero is returned if the given variable is not defined.

GETENVQQ searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes. (For example, the LIB environment variable defines the default search path for libraries to be linked with a program.)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: SETENVQQ, GETLASTERRORQQ

Example

```
!  Program to demonstrate GETENVQQ and SETENVQQ
USE DFLIB
INTEGER(4) lenv, lval
CHARACTER(80) env, val, enval

WRITE (*,900) ' Enter environment variable name to create, &
                modify, or delete: '
lenv = GETSTRQQ(env)
IF (lenv .EQ. 0) STOP
WRITE (*,900) ' Value of variable (ENTER to delete): '
lval = GETSTRQQ(val)
IF (lval .EQ. 0) val = ' '
enval = env(1:lenv) // '=' // val(1:lval)
IF (SETENVQQ(enval)) THEN
  lval = GETENVQQ(env(1:lenv), val)
  IF (lval .EQ. 0) THEN
    WRITE (*,*) 'Can''t get environment variable'
  ELSE IF (lval .GT. LEN(val)) THEN
    WRITE (*,*) 'Buffer too small'
  ELSE
    WRITE (*,*) env(:lenv), ': ', val(:lval)
    WRITE (*,*) 'Length: ', lval
  END IF
ELSE
  WRITE (*,*) 'Can''t set environment variable'
END IF
900  FORMAT (A, \)
END
```

GETEXITQQ

QuickWin Function: Gets the setting for a QuickWin application's exit behavior.

Module: USE DFLIB

Syntax

result = **GETEXITQQ** ( )

Results:

The result type is INTEGER(4). The result is exit mode with one of the following constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **QWIN$EXITPROMPT**: Displays a message box that reads "Program exited with exit status *n*. Exit Window?", where *n* is the exit status from the program. If the user chooses Yes, the application closes the window and terminates. If the user chooses No, the dialog box disappears and the user can manipulate the window as usual. The user must then close the

window manually.

- **QWIN$EXITNOPERSIST**: Terminates the application without displaying a message box.
- **QWIN$EXITPERSIST**: Leaves the application open without displaying a message box.

The default for both QuickWin and Console Graphics applications is QWIN$EXITPROMPT.

Compatibility

STANDARD GRAPHICS QUICKWIN.EXE LIB

See Also: SETEXITQQ, Using QuickWin

Example

```
!  Program to demonstrate GETEXITQQ
     USE DFLIB
     INTEGER i
     i = GETEXITQQ()
     SELECT CASE (i)
       CASE (QWIN$EXITPROMPT)
         WRITE(*, *) "Prompt on exit."
       CASE (QWIN$EXITNOPERSIST)
         WRITE(*,*) "Exit and close."
       CASE (QWIN$EXITPERSIST)
         WRITE(*,*) "Exit and leave open."
     END SELECT
     END
```

GETFILEINFOQQ

Run-Time Function: Returns information about the specified file. Filenames can contain wildcards (* and ?).

Module: USE DFLIB

Syntax

result = **GETFILEINFOQQ** (*files*, *buffer*, *handle*)

files
(Input) Character*(*). Search criteria. Can include a full path. Can include wildcards (* and ?).

buffer
(Output) Derived type file$info. Information about a file that matches the search criteria. The derived type file$info is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE file$info
  INTEGER(4) CREATION
  INTEGER(4) LASTWRITE
```

```
  INTEGER(4) LASTACCESS
  INTEGER(4) LENGTH
  INTEGER(4) PERMIT
  CHARACTER(255) NAME
END TYPE file$info
```

handle
(Input; output) INTEGER(4). Control mechanism. One of the following constants, defined in DFLIB.F90:

- **FILE$FIRST**: First matching file found.
- **FILE$LAST**: Previous file was the last valid file.
- **FILE$ERROR**: No matching file found.

Results:

The result type is INTEGER(4). The result is the nonblank length of the filename if a match was found, or 0 if no matching files were found.

To get information about one or more files, set handle to FILE$FIRST and call **GETFILEINFOQQ**. This will return information about the first file which matches the name and return a handle. If the program wants more files, it should call **GETFILEINFOQQ** with the handle. **GETFILEINFOQQ** must be called with the handle until **GETFILEINFOQQ** sets handle to FILE$LAST, or system resources may be lost.

The derived-type element variables FILE$INFO.CREATION, FILE$INFO.LASTWRITE, and FILE$INFO.LASTACCESS contain packed date and time information that indicates when the file was created, last written to, and last accessed, respectively. To break the time and date into component parts, call **UNPACKTIMEQQ**. FILE$INFO.LENGTH contains the length of the file in bytes. FILE$INFO.PERMIT contains a set of bit flags describing access information about the file as follows:

If this bit flag is set	The file is
FILE$ARCHIVE	Marked as having been copied to a backup device.
FILE$DIR	A subdirectory of the current directory. Each MS-DOS directory contains two special files, "." and "..". These are directory aliases created by MS-DOS for use in relative directory notation. The first refers to the current directory, and the second refers to the current directory’s parent directory.
FILE$HIDDEN	Hidden. It does not appear in the directory list you request from the command line, the Microsoft visual development environment browser, or File Manager.
FILE$READONLY	Write-protected. You can read the file, but you cannot make changes to it.
FILE$SYSTEM	Used by the operating system.
FILE$VOLUME	A logical volume, or partition, on a physical disk drive. This type of file appears only in the root directory of a physical device.

You can use the constant FILE$NORMAL to check that all bit flags are set to 0. If the derived-type element variable FILE$INFO.PERMIT is equal to FILE$NORMAL, the file has no special attributes. The variable FILE$INFO.NAME contains the short name of the file, not the full path of the file.

If an error occurs, call **GETLASTERRORQQ** to retrieve the error message, such as:

- **ERR$NOENT**: No directory entries matched the file specification.
- **ERR$NOENT**: Illegal filename specification.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: SETFILEACCESSQQ, SETFILETIMEQQ, UNPACKTIMEQQ

Example

```
 USE DFLIB
 CALL SHOWPERMISSION( )
 END
!  SUBROUTINE to demonstrate GETFILEINFOQQ
 SUBROUTINE SHOWPERMISSION()
 USE DFLIB
 CHARACTER(80) files

 INTEGER(4) handle, length
 CHARACTER(5) permit
 TYPE (FILE$INFO) info

WRITE (*, 900) ' Enter wildcard of files to view: '
900   FORMAT (A, \)
 length = GETSTRQQ(files)
 handle = FILE$FIRST
 DO WHILE (.TRUE.)
   length = GETFILEINFOQQ(files, info, handle)
   IF ((handle .EQ. FILE$LAST) .OR. &
     (handle .EQ. FILE$ERROR)) THEN
   SELECT CASE (GETLASTERRORQQ())
     CASE (ERR$NOMEM)
       WRITE (*,*) 'Out of memory'
     CASE (ERR$NOENT)
       EXIT
     CASE DEFAULT
       WRITE (*,*) 'Invalid file or path name'
   END SELECT
   END IF
   permit = ' '
   IF ((info.permit .AND. FILE$HIDDEN) .NE. 0) &
     permit(1:1) = 'H'
   IF ((info.permit .AND. FILE$SYSTEM) .NE. 0) &
     permit(2:2) = 'S'
   IF ((info.permit .AND. FILE$READONLY) .NE. 0) &
     permit(3:3) = 'R'
   IF ((info.permit .AND. FILE$ARCHIVE) .NE. 0) &
     permit(4:4) = 'A'
   IF ((info.permit .AND. FILE$DIR) .NE. 0) &
```

```
    permit(5:5) = 'D'
  WRITE (*, 9000) info.name, info.length, permit
9000  FORMAT (1X, A5, I9, ' ',A6)
END DO
END SUBROUTINE
```

GETFILLMASK

Graphics Subroutine: Returns the current pattern used to fill shapes.

Module: USE DFLIB

Syntax

CALL GETFILLMASK (*mask*)

mask
(Output) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element (byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of 0 are unchanged. The current graphics color is set with **SETCOLORRGB** or **SETCOLOR**. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with **SETFILLMASK**), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (**FLOODFILLRGB**, **PIE**, **ELLIPSE**, **POLYGON**, and **RECTANGLE**).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: ELLIPSE, FLOODFILLRGB, PIE, POLYGON, RECTANGLE, SETFILLMASK

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(1) style(8). array(8)
INTEGER(2) i
style = 0
style(1) = #F
style(3) = #F
style(5) = #F
style(7) = #F
CALL SETFILLMASK (style)
...
CALL GETFILLMASK (array)
WRITE (*, *) 'Fill mask in bits: '
DO i = 1, 8
```

```
  WRITE (*, '(B8)') array(i)
END DO
END
```

GETFONTINFO

Graphics Function: Gets the current font characteristics.

Module: USE DFLIB

Syntax

result = **GETFONTINFO** (*font*)

font
(Output) Derived type fontinfo. Set of characteristics of the current font. The fontinfo derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE fontinfo
  INTEGER(4) type        ! 1 = truetype, 0 = bit map
  INTEGER(4) ascent      ! Pixel distance from top to
                         ! baseline
  INTEGER(4) pixwidth    ! Character width in pixels,
                         ! 0=proportional
  INTEGER(4) pixheight   ! Character height in pixels
  INTEGER(4) avgwidth    ! Average character width in
                         ! pixels
  CHARACTER(32)xfacename ! Font name
  LOGICAL(1) italic      ! .TRUE. if current font
                         ! formatted italic
  LOGICAL(1) emphasized  ! .TRUE. if current font
                         ! formatted bold
  LOGICAL(1) underline   ! .TRUE. if current font
                         ! formatted underlined
END TYPE fontinfo
```

Results:

The result type is INTEGER(2). The result is zero if successful; otherwise, -1.

You must initialize fonts with **INITIALIZEFONTS** before calling any font-related function, including **GETFONTINFO**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETGTEXTEXTENT, GETGTEXTROTATION, GRSTATUS, OUTGTEXT, INITIALIZEFONTS, SETFONT, Using Fonts from the Graphics Library

Example

```
! Build as QuickWin or Standard Graphics
```

```
USE DFLIB
TYPE (FONTINFO) info
INTEGER(2) numfonts, return, line_spacing
numfonts = INITIALIZEFONTS ( )
return = GETFONTINFO(info)
line_spacing = info.pixheight + 2
END
```

GETGID

Portability Function: Retrieves the group ID of the user of a process.

Module: USE DFPORT

Syntax

result = **GETGID** ( )

Results:

The result type is INTEGER(4). The result is always 1.

This function is included for compatibility only.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
ISTAT = GETGID( )
```

GETGTEXTEXTENT

Graphics Function: Returns the width in pixels that would be required to print a given string of text (including any trailing blanks) with **OUTGTEXT** using the current font.

Module: USE DFLIB

Syntax

result = **GETGTEXTEXTENT** (*text*)

text
(Input) Character*(*). Text to be analyzed.

Results:

The result type is INTEGER(2). The result is the width of *text* in pixels if successful; otherwise, -1 (for example, if fonts have not been initialized with **INITIALIZEFONTS**).

This function is useful for determining the size of text that uses proportionally spaced fonts. You must initialize fonts with **INITIALIZEFONTS** before calling any font-related function, including **GETGTEXTEXTENT**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETFONTINFO, OUTGTEXT, SETFONT, INITIALIZEFONTS, GETGTEXTROTATION

Example

```
! Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(2) status, pwidth
CHARACTER(80) text
status= INITIALIZEFONTS( )
status= SETFONT('t''Arial''h22w10')
pwidth= GETGTEXTEXTENT('How many pixels wide is this?')
WRITE(*,*) pwidth
END
```

GETGTEXTROTATION

Graphics Function: Returns the current orientation of the font text output by **OUTGTEXT**.

Module: USE DFLIB

Syntax

result = **GETGTEXTROTATION** ( )

Results:

INTEGER(4). Current orientation of the font text output in tenths of degrees. Horizontal is 0°, and angles increase counterclockwise so that 900 tenths of degrees (90°) is straight up, 1800 tenths of degrees (180°) is upside-down and left, 2700 tenths of degrees (270°) is straight down, and so forth.

The orientation for text output with **OUTGTEXT** is set with **SETGTEXTROTATION**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: OUTGTEXT, SETFONT, SETGTEXTROTATION

Example

```
! Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER ang
REAL rang
ang = GETGTEXTROTATION( )
rang = FLOAT(ang)/10.0
WRITE(*,*) "Text tilt in degrees is: ", rang
END
```

GETHWNDQQ

QuickWin Function: Converts a window unit number into a Windows handle.

Module: USE DFLIB

Syntax

result = **GETHWNDQQ** (*unit*)

unit
(Input) INTEGER(4). Window unit number. If *unit* is set to QWIN$FRAMEWINDOW (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory), the handle of the frame window is returned.

Results:

The result type is INTEGER(4). The result is a true Windows handle to the window. It returns -1 if *unit* is not open.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: GETACTIVEQQ, GETUNITQQ, SETACTIVEQQ Using QuickWin

GETIMAGE, GETIMAGE_W

Graphics Subroutine: Stores the screen image defined by a specified bounding rectangle.

Module: USE DFLIB

Syntax

CALL GETIMAGE (*x1*, *y1*, *x2*, *y2*, *image*)
CALL GETIMAGE_W (*wx1*, *wy1*, *wx2*, *wy2*, *image*)

x1, *y1*

(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

x2, y2
(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

wx1, wy1
(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.

wx2, wy2
(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

image
(Output) INTEGER(1). Array of single-byte integers. Stored image buffer.

GETIMAGE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). **GETIMAGE_W** defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

The buffer used to store the image must be large enough to hold it. You can determine the image size by calling **IMAGESIZE** at run time, or by using the formula described under **IMAGESIZE**. After you have determined the image size, you can dimension the buffer accordingly.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: IMAGESIZE, PUTIMAGE

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(1), ALLOCATABLE:: buffer (:)
INTEGER(2) status, x, y, error
INTEGER(4) imsize
x = 50
y = 30
status = ELLIPSE ($GFILLINTERIOR, INT2(x-15), &
                  INT2(y-15), INT2( x+15), INT2(y+15))
imsize = IMAGESIZE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16))
ALLOCATE(buffer (imsize), STAT = error)
IF (error .NE. 0) THEN
  STOP 'ERROR: Insufficient memory'
END IF
CALL GETIMAGE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16), buffer)
END
```

GETLASTERRORQQ

Run-Time Function: Returns the last error set by a run-time procedure.

Module: USE DFLIB

Syntax

result = **GETLASTERRORQQ ( )**

Results:

The result type is INTEGER(4). The result is the most recent error code generated by a run-time procedure.

Run-time functions that return a logical or integer value sometimes also provide an error code that identifies the cause of errors. **GETLASTERRORQQ** retrieves the most recent error message. The error constants are in DFLIB.F90 (in the \DF98\INCLUDE subdirectory). The following table shows the run-time library routines and the errors each routine produces:

This run-time routine	Produces these errors
RUNQQ	**ERR$NOMEM, ERR$2BIG, ERR$INVAL, ERR$NOENT, ERR$NOEXEC**
SYSTEMQQ	**ERR$NOMEM, ERR$2BIG, ERR$NOENT, ERR$NOEXEC**
GETDRIVESIZEQQ	**ERR$INVAL, ERR$NOENT**
GETDRIVESQQ	no error
GETDRIVEDIRQQ	**ERR$NOMEM, ERR$RANGE**
CHANGEDRIVEQQ	**ERR$INVAL, ERR$NOENT**
CHANGEDIRQQ	**ERR$NOMEM, ERR$NOENT**
MAKEDIRQQ	**ERR$NOMEM, ERR$ACCES, ERR$NOENT**
DELDIRQQ	**ERR$NOMEM, ERR$ACCES, ERR$NOENT**
FULLPATHQQ	**ERR$NOMEM, ERR$INVAL**
SPLITPATHQQ	**ERR$NOMEM, ERR$INVAL**
GETFILEINFOQQ	**ERR$NOMEM, ERR$NOENT, ERR$INVAL**
SETFILETIMEQQ	**ERR$NOMEM, ERR$ACCES, ERR$INVAL, ERR$MFILE, ERR$NOENT**
SETFILEACCESSQQ	**ERR$NOMEM, ERR$INVAL, ERR$ACCES**
DELFILESQQ	**ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$INVAL**
RENAMEFILEQQ	**ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$XDEV**

FINDFILEQQ	**ERR$NOMEM, ERR$NOENT**
PACKTIMEQQ	no error
UNPACKTIMEQQ	no error
COMMITQQ	**ERR$BADF**
GETCHARQQ	no error
PEEKCHARQQ	no error
GETSTRQQ	no error
GETLASTERRORQQ	no error
SETERRORMODEQQ	no error
GETENVQQ	**ERR$NOMEM, ERR$NOENT**
SETENVQQ	**ERR$NOMEM, ERR$INVAL**
SLEEPQQ	no error
BEEPQQ	no error
SORTQQ	**ERR$INVAL**
BSEARCHQQ	**ERR$INVAL**

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

GETLINESTYLE

Graphics Function: Returns the current graphics line style.

Module: USE DFLIB

Syntax

result = **GETLINESTYLE** ( )

Results:

The result type is INTEGER(2). The result is the current line style.

GETLINESTYLE retrieves the mask (line style) used for line drawing. The mask is a 16-bit number, where each bit represents a pixel in the line being drawn.

If a bit is 1, the corresponding pixel is colored according to the current graphics color and logical write mode; if a bit is 0, the corresponding pixel is left unchanged. The mask is repeated for the entire length of the line. The default mask is #FFFF (a solid line). A dashed line can be represented by #FF00 (long dashes) or #F0F0 (short dashes).

The line style is set with **SETLINESTYLE**. The current graphics color is set with **SETCOLORRGB** or **SETCOLOR**. **SETWRITEMODE** affects how the line is displayed.

The line style retrieved by **GETLINESTYLE** affects the drawing of straight lines as in **LINETO, POLYGON** and **RECTANGLE**, but not the drawing of curved lines as in **ARC**, **ELLIPSE** or **PIE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: LINETO, POLYGON, RECTANGLE, SETCOLORRGB, SETFILLMASK, SETLINESTYLE, SETWRITEMODE

Example

```
!  Build as Graphics
     USE DFLIB
     INTEGER(2) lstyle

     lstyle = GETLINESTYLE()
     WRITE (*, 100) lstyle, lstyle
100  FORMAT (1X, 'Line mask in Hex ', Z4, ' and binary ', B16)
     END
```

GETLOG

Portability Subroutine: Retrieves the user's login name.

Module: USE DFPORT

Syntax

CALL GETLOG (*name*)

name
(Output) Character*(*). User's login name, or all blanks if the name cannot be determined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use dfport
character*20 username
```

```
CALL GETLOG (username)
print *, "You logged in as ",username
```

GETPHYSCOORD

Graphics Subroutine: Translates viewport coordinates to physical coordinates.

Module: USE DFLIB

Syntax

CALL GETPHYSCOORD (*x*, *y*, *t*)

x, *y*
(Input) INTEGER(2). Viewport coordinates to be translated to physical coordinates.

t
(Output) Derived Type xycoord. Physical coordinates of the input viewport position. The *xycoord* derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with **SETVIEWPORT**. Both take integer coordinate values. Window coordinates refer to a window sized with **SETWINDOW** or **SETWSIZEQQ**. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see Understanding Coordinate Systems in the *Programmer's Guide*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETVIEWCOORD, GETWINDOWCOORD, SETCLIPRGN, SETVIEWPORT

Example

```
!  Program to demonstrate GETPHYSCOORD, GETVIEWCOORD
!  and GETWINDOWCOORD. Build as QuickWin or Standard
!  Graphics
USE DFLIB
TYPE (xycoord) viewxy, physxy
TYPE (wxycoord) windxy
CALL SETVIEWPORT(INT2(80), INT2(50), &
                 INT2(240), INT2(150))
!  Get viewport equivalent of point (100, 90)
CALL GETVIEWCOORD (INT2(100), INT2(90), viewxy)
!  Get physical equivalent of viewport coordinates
CALL GETPHYSCOORD (viewxy.xcoord, viewxy.ycoord, &
                   physxy)
```

```
!  Get physical equivalent of viewport coordinates
CALL GETWINDOWCOORD (viewxy.xcoord, viewxy.ycoord, &
                     windxy)

!  Write viewport coordinates
WRITE (*,*) viewxy.xcoord, viewxy.ycoord
!  Write physical coordinates
WRITE (*,*) physxy.xcoord, physxy.ycoord
!  Write window coordinates
WRITE (*,*) windxy.wx, windxy.wy
END
```

GETPID

Portability Function: Returns the process ID of the current process.

Module: USE DFPORT

Syntax

result = **GETPID** ( )

Results:

The result type is INTEGER(4). The result is the process ID number of the current process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFPORT
INTEGER(4) istat
istat = GETPID()
```

GETPIXEL, GETPIXEL_W

Graphics Function: Returns the color index of the pixel at a specified location.

Module: USE DFLIB

Syntax

result = **GETPIXEL** (*x*, *y*)
result = **GETPIXEL_W** (*wx*, *wy*)

x, *y*
(Input) INTEGER(2). Viewport coordinates for pixel position.

wx, wy
(Input) REAL(8). Window coordinates for pixel position.

Results:

The result type is INTEGER(2). The result is the pixel color index if successful; otherwise, -1 (if the pixel lies outside the clipping region, for example).

Color routines without the **RGB** suffix, such as **GETPIXEL**, use color indexes, not true color values, and limit you to colors in the palette, at most 256. To access all system colors, use **SETPIXELRGB** to specify an explicit Red-Green-Blue value and retrieve the value with **GETPIXELRGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPIXELRGB, GRSTATUS, REMAPALLPALETTERGB, REMAPPALETTERGB, SETCOLOR, GETPIXELS, SETPIXEL

GETPIXELRGB, GETPIXELRGB_W

Graphics Function: Returns the Red-Green-Blue (RGB) color value of the pixel at a specified location.

Module: USE DFLIB

Syntax

result = **GETPIXELRGB** (*x, y*)
result = **GETPIXELRGB_W** (*wx, wy*)

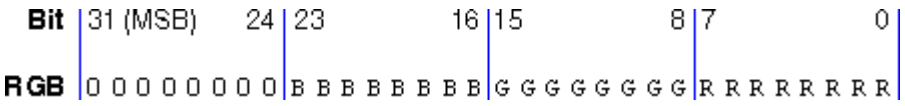
x, y
(Input) INTEGER(2). Viewport coordinates for pixel position.

wx, wy
(Input) REAL(8). Window coordinates for pixel position.

Results:

The result type is INTEGER(4). The result is the pixel's current RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with **GETPIXELRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETPIXELRGB returns the true color value of the pixel, set with **SETPIXELRGB**, **SETCOLORRGB**, **SETBKCOLORRGB**, or **SETTEXTCOLORRGB**, depending on the pixel's position and the current configuration of the screen.

SETPIXELRGB (and the other RGB color selection functions **SETCOLORRGB**, **SETBKCOLORRGB**, and **SETTEXTCOLORRGB**) sets colors to a color value chosen from the entire available range. The non-RGB color functions (**SETPIXELS, SETCOLOR, SETBKCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETPIXELRGB, GETPIXELSRGB, SETPIXELSRGB, GETPIXEL, GETPIXEL_W

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(4) pixcolor, rseed
INTEGER(2) status
REAL rnd1, rnd2
LOGICAL(4) winstat
TYPE (windowconfig) wc
CALL GETTIM (status, status, status, INT2(rseed))
CALL SEED (rseed)
CALL RANDOM (rnd1)
CALL RANDOM (rnd2)
!  Get the color index of a random pixel, normalized to
!  be in the window. Then set current color to that
!  pixel color.
winstat = GETWINDOWCONFIG(wc)
xnum = wc.numxpixels
ynum = wc.numypixels
pixcolor = GETPIXELRGB( INT2( rnd1*xnum ), INT2( rnd2*ynum ))
status = SETCOLORRGB (pixcolor)
END
```

GETPIXELS

Graphics Subroutine: Gets the color indexes of multiple pixels.

Module: USE DFLIB

Syntax

CALL GETPIXELS (*n*, *x*, *y*, *color*)

n
(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other arguments.

x, *y*
(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to get.

color
(Output) INTEGER(2). Array to be filled with the color indexes of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with color indexes of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If the pixel is outside the clipping region, the value placed in the *color* array is undefined. Calls to **GETPIXELS** with *n* less than 1 are ignored. **GETPIXELS** is a much faster way to acquire multiple pixel color indexes than individual calls to **GETPIXEL**.

The range of possible pixel color index values is determined by the current video mode and palette, at most 256 colors. To access all system colors you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function such as **SETPIXELSRGB** and retrieve the value with **GETPIXELSRGB**, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPIXELSRGB, SETPIXELSRGB, GETPIXEL, SETPIXELS

GETPIXELSRGB

Graphics Subroutine: Returns the Red-Green-Blue (RGB) color values of multiple pixels.

Module: USE DFLIB

Syntax

CALL GETPIXELSRGB (*n*, *x*, *y*, *color*)

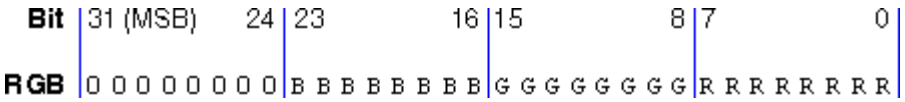
n
(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other argument arrays.

x, y
(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels.

color
(Output) INTEGER(4). Array to be filled with RGB color values of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with the RGB color values of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the values you retrieve with **GETPIXELSRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETPIXELSRGB is a much faster way to acquire multiple pixel RGB colors than individual calls to **GETPIXELRGB. GETPIXELSRGB** returns an array of true color values of multiple pixels, set with **SETPIXELSRGB**, **SETCOLORRGB**, **SETBKCOLORRGB**, or **SETTEXTCOLORRGB**, depending on the pixels' positions and the current configuration of the screen.

SETPIXELSRGB (and the other RGB color selection functions **SETCOLORRGB**, **SETBKCOLORRGB**, and **SETTEXTCOLORRGB**) sets colors to a color value chosen from the entire available range. The non-RGB color functions (**SETPIXELS, SETCOLOR, SETBKCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETPIXELSRGB, GETPIXELRGB, GETPIXELRGB_W, GETPIXELS, SETPIXELS

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(4) color(50), result
INTEGER(2) x(50), y(50), status
TYPE (xycoord) pos

result = SETCOLORRGB(#FF)
CALL MOVETO(INT2(0), INT2(0), pos)
status = LINETO(INT2(100), INT2(200))

!  Get 50 pixels at line 30 in viewport
DO i = 1, 50
  x(i) = i-1
  y(i) = 30
END DO
CALL GETPIXELSRGB(300, x, y, color)
!  Move down 30 pixels and redisplay pixels
DO i = 1, 50
  y(i) = y(i) + 30
END DO
CALL SETPIXELSRGB (50, x, y, color)
END
```

GETSTATUSFPQQ (x86 only)

Run-Time Subroutine: Returns the floating-point processor status word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL GETSTATUSFPQQ (*status*)

status
(Output) INTEGER(2). Floating-point processor status word.

The floating-point status word shows whether various floating-point exception conditions have occurred. Visual Fortran initially clears (sets to 0) all status flags, but after an exception occurs it does not reset the flags before performing additional floating-point operations. A status flag with a value of one thus shows there has been at least one occurence of the corresponding exception. The following table lists the status flags and their values:

Parameter name	Hex value	Description
FPSW$MSW_EM	#003F	Status Mask (set all flags to 1)
FPSW$INVALID	#0001	An invalid result occurred
FPSW$DENORMAL	#0002	A denormal (very small number) occurred
FPSW$ZERODIVIDE	#0004	A divide by zero occurred

FPSW$OVERFLOW	#0008	An overflow occurred
FPSW$UNDERFLOW	#0010	An underflow occurred
FPSW$INEXACT	#0020	Inexact precision occurred

You can use a logical comparison on the status word returned by **GETSTATUSFPQQ** to determine which of the six floating-point exceptions listed in the table has occurred.

An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0. By default, the denormal, underflow and inexact precision exceptions are disabled, and the invalid, overflow and divide-by-zero exceptions are enabled. Exceptions can be enabled and disabled by clearing and setting the flags with **SETCONTROLFPQQ**. You can use **GETCONTROLFPQQ** to determine which exceptions are currently enabled and disabled.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues. You can find out which exceptions (if any) occurred by calling **GETSTATUSFPQQ**.

If errors on floating-point exceptions are enabled (by clearing the flags to 0 with **SETCONTROLFPQQ**), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with **SIGNALQQ** and branch to your own error-handling routines.

For a full discussion of the floating-point status word, exceptions, and error handling, see The Floating-Point Environment in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: SETCONTROLFPQQ, GETCONTROLFPQQ, SIGNALQQ, MATHERRQQ

Example

```
!  Program to demonstrate GETSTATUSFPQQ
 USE DFLIB
 INTEGER(2) status

 CALL GETSTATUSFPQQ(status)
!  check for divide by zero
 IF (IAND(status, FPSW$ZERODIVIDE) .NE. 0) THEN
   WRITE (*,*) 'Divide by zero occurred. Look    &
   for NaN or signed infinity in resultant data.'
 END IF
END
```

GETSTRQQ

Run-Time Function: Reads a character string from the keyboard using buffered input.

Module: USE DFLIB

Syntax

result = **GETSTRQQ** (*buffer*)

buffer
(Output) Character*(*). Character string returned from keyboard, padded with blanks.

Results:

The result type is INTEGER(4). The result is the number of characters placed in *buffer*.

The function does not complete until the user presses RETURN or ENTER.

Using **GETSTRQQ** rather than **READ** allows you to use MS-DOS command-line editing, or command-line recall if a recall utility such as DOSKEY is active.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: READ, GETCHARQQ, PEEKCHARQQ

Example

```
!  Program to demonstrate GETSTRQQ
 USE DFLIB
 INTEGER(4) length, result
 CHARACTER(80) prog, args
 WRITE (*, '(A, \)') ' Enter program to run: '
 length = GETSTRQQ (prog)
 WRITE (*, '(A, \)') ' Enter arguments: '
 length = GETSTRQQ (args)
 result = RUNQQ (prog, args)
 IF (result .EQ. -1) THEN
   WRITE (*,*) 'Couldn''t run program'
 ELSE
   WRITE (*, '(A, Z4, A)') 'Return code : ', result, 'h'
 END IF
 END
```

GETTEXTCOLOR

Graphics Function: Gets the current text color index.

Module: USE DFLIB

Syntax

result = **GETTEXTCOLOR** ( )

Results:

The result type is INTEGER(2). It is the current text color index.

GETTEXTCOLOR returns the text color index set by **SETTEXTCOLOR**. **SETTEXTCOLOR** affects text output with **OUTTEXT**, **WRITE**, and **PRINT**. The background color index is set with **SETBKCOLOR** and returned with **GETBKCOLOR**. The color index of graphics over the background color is set with **SETCOLOR** and returned with **GETCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use **SETTEXTCOLORRGB**, **SETBKCOLORRGB**, and **SETCOLORRGB**.

The default text color index is 15, which is associated with white unless the user remaps the palette.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: OUTTEXT, REMAPPALETTERGB, SETCOLOR, SETTEXTCOLOR

GETTEXTCOLORRGB

Graphics Function: Gets the Red-Green-Blue (RGB) value of the current text color (used with **OUTTEXT**, **WRITE** and **PRINT**).

Module: USE DFLIB

Syntax

result = **GETTEXTCOLORRGB** ( )

Results:

The result type is INTEGER(4). It is the RGB value of the current text color.

In each RGB color calue, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with **GETTEXTCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Bit	31 (MSB) 24	23 16	15 8	7 0
RGB	0 0 0 0 0 0 0 0	B B B B B B B B	G G G G G G G G	R R R R R R R R

Larger numbers correspond to stronger color intensity with binary (hex FF) the maximum for each of

the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as **OUTTEXT**, **WRITE**, and **PRINT**), set with **SETTEXTCOLORRGB**. The RGB color value used for graphics is set and returned with **SETCOLORRGB** and **GETCOLORRGB**. **SETCOLORRGB** controls the color used by the graphics function **OUTGTEXT**, while **SETTEXTCOLORRGB** controls the color used by all other text output functions. The RGB background color value for both text and graphics is set and returned with **SETBKCOLORRGB** and **GETBKCOLORRGB**.

SETTEXTCOLORRGB (and the other RGB color selection functions **SETBKCOLORRGB**, and **SETCOLORRGB**) sets the color to a color value chosen from the entire available range. The non-RGB color functions (**SETTEXTCOLOR**, **SETBKCOLOR**, and **SETCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETTEXTCOLORRGB, GETBKCOLORRGB, GETCOLORRGB, GETTEXTCOLOR

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(4) oldtextc, oldbackc, temp
TYPE (rccoord) curpos
!  Save color settings
oldtextc = GETTEXTCOLORRGB()
oldbackc = GETBKCOLORRGB()
CALL CLEARSCREEN( $GCLEARSCREEN )
!  Reset colors
temp = SETTEXTCOLORRGB(#00FFFF)  ! full red + full green
                                 ! = full yellow text
temp = SETBKCOLORRGB(#FF0000)    ! blue background
CALL SETTEXTPOSITION( INT2(4), INT2(15), curpos)
CALL OUTTEXT( 'Hello, world')
!  Restore colors
temp = SETTEXTCOLORRGB(oldtextc)
temp = SETBKCOLORRGB(oldbackc)
END
```

GETTEXTPOSITION

Graphics Subroutine: Returns the current text position.

Module: USE DFLIB

Syntax

CALL GETTEXTPOSITION (*t*)

t
(Output) Derived type rccord. Current text position.The derived type rccoord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE rccoord
  INTEGER(2) row   ! Row coordinate
  INTEGER(2) col   ! Column coordinate
END TYPE rccoord
```

The text position given by coordinates (1, 1) is defined as the upper-left corner of the text window. Text output from the **OUTTEXT** function (and **WRITE** and **PRINT** statements) begins at the current text position. Font text is not affected by the current text position. Graphics output, including **OUTGTEXT** output, begins at the current graphics output position, which is a separate position returned by **GETCURRENTPOSITION**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETTEXTPOSITION, GETCURRENTPOSITION, OUTTEXT, WRITE, SETTEXTWINDOW

Example

```
!  Build as QuickWin or Standard Graphics
     USE DFLIB
     TYPE (rccoord) textpos
     CALL GETTEXTPOSITION (textpos)
     END
```

GETTEXTWINDOW

Graphics Subroutine: Finds the boundaries of the current text window.

Module: USE DFLIB

Syntax

CALL GETTEXTWINDOW (*r1*, *c1*, *r2*, *c2*)

r1, *c1*
(Output) INTEGER(2). Row and column coordinates for upper-left corner of the text window.

r2, *c2*
(Output) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

Output from **OUTTEXT** and **WRITE** is limited to the text window. By default, this is the entire window, unless the text window is redefined by **SETTEXTWINDOW**.

The window defined by **SETTEXTWINDOW** has no effect on output from **OUTGTEXT**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETTEXTPOSITION, OUTTEXT, WRITE, SCROLLTEXTWINDOW, SETTEXTPOSITION, SETTEXTWINDOW, WRAPON

Example

```
!  Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(2) top, left, bottom, right
DO i = 1, 10
  WRITE(*,*) "Hello, world"
END DO
!  Save text window position
 CALL GETTEXTWINDOW (top, left, bottom, right)
!  Scroll text window down seven lines
 CALL SCROLLTEXTWINDOW (INT2(-7))
!  Restore text window
 CALL SETTEXTWINDOW (top, left, bottom, right)
 WRITE(*,*) "At beginning again"
 END
```

GETTIM

Run-Time Subroutine: Returns the time.

Module: USE DFLIB

Syntax

CALL GETTIM (*ihr*, *imin*, *isec*, *i100th*)

ihr
(Output) INTEGER(2). Hour (0-23).

imin
(Output) INTEGER(2). Minute (0-59).

isec
(Output) INTEGER(2). Second (0-59).

i100th

(Output) INTEGER(2). Hundredths of a second (0-99).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: GETDAT, SETDAT, SETTIM, CLOCK, CTIME, DTIME, ETIME, GMTIME, ITIME, LTIME, RTC, SECNDS, TIME, TIMEF

Example

See the example in GETDAT.

GETUID

Portability Function: Retrieves the user ID of the calling process.

Module: USE DFPORT

Syntax

result = **GETUID** ( )

Results:

The result type is INTEGER(4). The result is always 1.

This function is included for compatibility only.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFPORT
integer(4) istat
ISTAT = GETUID( )
```

GETUNITQQ

QuickWin Function: Returns the unit number corresponding to the specified Windows handle.

Module: USE DFLIB

Syntax

result = **GETUNITQQ** (*whandle*)

whandle
(Input) INTEGER(4). The Windows handle to the window; this is a unique ID.

Results:

The result type is INTEGER(4). The result is the unit number corresponding to the specified Windows handle. It returns -1 if *whandle* does not exist.

This routine is the inverse of **GETHWNDQQ**.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: GETHWNDQQ, Using QuickWin

GETVIEWCOORD, GETVIEWCOORD_W

Graphics Subroutine: Translates physical coordinates or window coordinates to viewport coordinates.

Module: USE DFLIB

Syntax

CALL GETVIEWCOORD (*x*, *y*, *t*)
CALL GETVIEWCOORD_W (*wx*, *wy*, *wt*)

x, *y*
(Input) INTEGER(2). Physical coordinates to be converted to viewport coordinates.

t
(Output) Derived type xycoord. Viewport coordinates. The xycoord derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

wx, *wy*
(Input) REAL(8). Window coordinates to be converted to viewport coordinates.

wt
(Output) Derived type xycoord. Viewport coordinates. The xycoord derived type is defined in DFLIB.F90 (see above).

Viewport coordinates refer to an area of the screen defined as the viewport with **SETVIEWPORT**.

Physical coordinates refer to the whole screen. Both take integer coordinate values. Window coordinates refer to a window sized with **SETWINDOW** or **SETWSIZEQQ**. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see Understanding Coordinate Systems in the *Programmer's Guide*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPHYSCOORD, GETWINDOWCOORD

Example

See the example program in GETPHYSCOORD.

GETWINDOWCONFIG

QuickWin Function: Gets the properties of the current window.

Module: USE DFLIB

Syntax

result = **GETWINDOWCONFIG** (*wc*)

wc
(Output) Derived type windowconfig. Contains window properties. The windowconfig derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE windowconfig
INTEGER(2) numxpixels    ! Number of pixels on x-axis
INTEGER(2) numypixels    ! Number of pixels on y-axis
INTEGER(2) numtextcols   ! Number of text columns
                         ! available
INTEGER(2) numtextrows   ! Number of text rows
                         ! available
INTEGER(2) numcolors     ! Number of color indexes
INTEGER(4) fontsize      ! Size of default font. Set
               ! to QWIN$EXTENDFONT when using
               ! multibyte characters, in which case
               ! extendfontsize sets the font size.
CHARACTER(80) title      ! window title
INTEGER(2) bitsperpixel  ! number of bits per pixel
! The next three parameters support multibyte character
! sets (such as Japanese)
CHARACTER(32) extendfontname     ! any nonproportionally
               ! spaced font available on the system
INTEGER(4) extendfontsize        ! takes same values as
               ! fontsize, but used for multibyte
               ! character sets when fontsize set to
               ! QWIN$EXTENDFONT
INTEGER(4) extendfontattributes  ! font attributes
               ! such as bold and italic for
               ! multibyte character sets
```

```
    END TYPE windowconfig
```

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE. (for example, if there is no active child window).

GETWINDOWCONFIG returns information about the active child window. If you have not set the window properties with **SETWINDOWCONFIG**, **GETWINDOWCONFIG** returns default window values.

A typical set of values would be 1024 *x* pixels, 768 *y* pixels, 128 text columns, 48 text rows, and a font size of 8x16 pixels. The resolution of the display and the assumed font size of 8x16 pixels generates the number of text rows and text columns. The resolution (in this case, 1024 *x* pixels by 768 *y* pixels) is the size of the *virtual* window. To get the size of the *physical* window visible on the screen, use **GETWSIZEQQ**. In this case, **GETWSIZEQQ** returned the following values: (0,0) for the x and y position of the physical window, 25 for the height or number of rows, and 71 for the width or number of columns.

The number of colors returned depends on the video drive. The window title defaults to "Graphic1" for the default window. All of these values can be changed with **SETWINDOWCONFIG**.

Note that the bitsperpixel field in the windowconfig derived type is an output field only, while the other fields return output values to **GETWINDOWCONFIG** and accept input values from **SETWINDOWCONFIG**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETWSIZEQQ SETWINDOWCONFIG, SETACTIVEQQ, Using QuickWin

Example

```
!Build as QuickWin or Standard Graphics App.
USE DFLIB
LOGICAL(4) status
TYPE (windowconfig) wc
status = GETWINDOWCONFIG(wc)
IF(wc.numtextrows .LT. 10) THEN
  wc.numtextrows = 10
  status = SETWINDOWCONFIG(wc)
  IF(.NOT. status ) THEN  ! if setwindowconfig error
    status = SETWINDOWCONFIG(wc)   ! reset
             ! setwindowconfig with corrected values
    status = GETWINDOWCONFIG(wc)
    IF(wc.numtextrows .NE. 10) THEN
       WRITE(*,*) 'Error: Cannot increase text rows to 10'
    END IF
  END IF
END IF
END
```

GETWINDOWCOORD

Graphics Subroutine: Translates viewport coordinates to window coordinates.

Module: USE DFLIB

Syntax

CALL GETWINDOWCOORD (*x*, *y*, *wt*)

x, *y*
(Input) INTEGER(2). Viewport coordinates to be converted to window coordinates.

wt
(Output) Derived type wxycoord. Window coordinates. The wxycoord derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE wxycoord
  REAL(8) wx   ! x-coordinate
  REAL(8) wy   ! y-coordinate
END TYPE wxycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with **SETVIEWPORT**. Both take integer coordinate values. Window coordinates refer to a window sized with **SETWINDOW** or **SETWSIZEQQ**. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see Understanding Coordinate Systems in the *Programmer's Guide*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCURRENTPOSITION, GETPHYSCOORD, GETVIEWCOORD, MOVETO, SETVIEWPORT, SETWINDOW

Example

See the example program in GETPHYSCOORD.

GETWRITEMODE

Graphics Function: Returns the current logical write mode, which is used when drawing lines with the **LINETO, POLYGON**, and **RECTANGLE** functions.

Module: USE DFLIB

Syntax

result = **GETWRITEMODE** ( )

Results:

The result type is INTEGER(2). The result is the current write mode. Possible return values are:

- **$GPSET**: Causes lines to be drawn in the current graphics color. (default)
- **$GAND**: Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- **$GOR**: Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- **$GPRESET**: Causes lines to be drawn in the color that is the logical NOT of the current graphics color.
- **$GXOR**: Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

The default value is $GPSET. These constants are defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory).

The write mode is set with **SETWRITEMODE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETWRITEMODE, SETLINESTYLE, LINETO, POLYGON, PUTIMAGE, RECTANGLE, SETCOLORRGB, SETFILLMASK, GRSTATUS

Example

```
! Build as QuickWin or Standard Graphics App.
   USE DFLIB
   INTEGER(2) mode
   mode = GETWRITEMODE()
   END
```

GETWSIZEQQ

QuickWin Function: Gets the size and position of a window.

Module: USE DFLIB

Syntax

result = **GETWSIZEQQ** (*unit, ireq, winfo*)

unit
(Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5 and 6 refer to the default startup window only if you have not explicitly opened them with the **OPEN** statement. To access information about the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW, defined in DFLIB.F90 in the \DF98 \INCLUDE subdirectory.

ireq
(Input) INTEGER(4). Specifies what information is obtained. The following symbolic constants, defined in DFLIB.F90, are available:

- **QWIN$SIZEMAX**: Gets information about the maximum window size.
- **QWIN$SIZECURR**: Gets information about the current window size.

winfo
(Output) Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type qwinfo is defined in DFLIB.F90 as follows:

```
TYPE QWINFO
  INTEGER(2) TYPE  ! request type (controls
                   ! SETWSIZEQQ)
  INTEGER(2) X     ! x coordinate for upper left
  INTEGER(2) Y     ! y coordinate for upper left
  INTEGER(2) H     ! window height
  INTEGER(2) W     ! window width
END TYPE QWINFO
```

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width returned for a frame window reflects the size in pixels of the client area *excluding* any borders, menus, and status bar at the bottom of the frame window. You should adjust the values used in **SETWSIZEQQ** to take this into account.

The client area is the area actually available to place child windows.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: GETWINDOWCONFIG, SETWSIZEQQ, Using QuickWin

GMTIME

Portability Subroutine: Returns the Greenwich mean time in an array of time elements.

Module: USE DFPORT

Syntax

CALL GMTIME (*stime*, *tarray*)

stime
(Input) Default integer (INTEGER(4) unless changed by the user). Numeric time data to be formatted. Number of seconds since 00:00:00 Greenwich mean time, January1970.

tarray
(Output) Default integer (INTEGER(4) unless changed by the user). One-dimensional array with 9 elements used to contain numeric time data. The elements of *tarray* are returned as follows:

Element	Value
tarray(1)	Seconds (0-59)
tarray(2)	Minutes (0-59)
tarray(3)	Hours (0-23)
tarray(4)	Day of month (1-31)
tarray(5)	Month (0-11)
tarray(6)	Year number in century (0-99)
tarray(7)	Day of week (0-6, where 0 is Sunday)
tarray(8)	Day of year (0-365)
tarray(9)	Daylight saving flag (0 if standard time, 1 if daylight saving time)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
use dfport
integer(4) stime, timearray(9)
```

```
CALL GMTIME (stime, timearray)
print *, timearray
```

GOTO -- Assigned

Statement: (Obsolescent) Transfers control to the statement whose label was most recently assigned to a variable.

Syntax

GOTO *var* [[**,**] (*label-list*)]

var
Is a scalar integer variable.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Obsolescent Features in Fortran 90, GOTO -- Computed GOTO, GOTO -- Unconditional GOTO, Execution Control

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.

The following shows another example:

```
    ASSIGN 10 TO n
    GOTO n
10  CONTINUE
```

The following example uses an assigned **GOTO** statement to check the value of view:

```
C   Show user appropriate view of data depending on
C   security clearance.
    GOTO view (100, 200, 400)
```

GOTO -- Computed

Statement: Transfers control to one of a set of labeled branch target statements based on the value of an expression.

Syntax

GOTO (*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.

Compatibility

Console Standard Graphics QuickWin Graphics Windows DLL LIB

See Also: GOTO -- Unconditional GOTO, Execution Control

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
```

The following shows another example:

```
   next = 1
C
C  The following statement transfers control to statement 10:
C
   GOTO (10, 20) next
   ...
10 CONTINUE
   ...
20 CONTINUE
```

GOTO -- Unconditional

Statement: Transfers control to the same branch target statement every time it executes.

Syntax

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.

Compatibility

Console Standard Graphics QuickWin Graphics Windows DLL LIB

See Also: GOTO -- Computed GOTO, Execution Control

Examples

The following are examples of **GO TO** statements:

```
  GO TO 7734
  GO TO 99999
```

The following shows another example:

```
    integer(2) in
10  print *, 'enter a number from one to ten: '
    read *, in
    select case (in)
    case (1:10)
      exit
    case default
      print *, 'wrong entry, try again'
      goto 10
    end select
```

GRSTATUS

Graphics Function: Returns the status of the most recently used graphics routine.

Module: USE DFLIB

Syntax

result = **GRSTATUS ( )**

Results:

The result type is INTEGER(2). The result is the status of the most recently used graphics function.

Use **GRSTATUS** immediately following a call to a graphics routine to determine if errors or warnings were generated. Return values less than 0 are errors, and values greater than 0 are warnings.

The following symbolic constants are defined in the DFLIB.F90 module file (in the \DF98\INCLUDE subdirectory) for use with **GRSTATUS**:

Constant	Meaning
$GRFILEWRITEERROR	Error writing bitmap file
$GRFILEOPENERROR	Error opening bitmap file
$GRIMAGEREADERROR	Error reading image
$GRBITMAPDISPLAYERROR	Error displaying bitmap
$GRBITMAPTOOLARGE	Bitmap too large
$GRIMPROPERBITMAPFORMAT	Improper format for bitmap file

$GRFILEREADERROR	Error reading file
$GRNOBITMAPFILE	No bitmap file
$GRINVALIDIMAGEBUFFER	Image buffer data inconsistent
$GRINSUFFICIENTMEMORY	Not enough memory to allocate buffer or to complete a fill operation
$GRINVALIDPARAMETER	One or more parameters invalid
$GRMODENOTSUPPORTED	Requested video mode not supported
$GRERROR	Graphics error
$GROK	Success
$GRNOOUTPUT	No action taken
$GRCLIPPED	Output was clipped to viewport
$GRPARAMETERALTERED	One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order

After a graphics call, compare the return value of **GRSTATUS** to $GROK. to determine if an error has occurred. For example:

```
IF ( GRSTATUS .LT. $GROK ) THEN
! Code to handle graphics error goes here
ENDIF
```

The following routines cannot give errors, and they all set **GRSTATUS** to $GROK:

DISPLAYCURSOR	**GETTEXTCOLORRGB**
GETBKCOLOR	**GETTEXTPOSITION**
GETBKCOLORRGB	**GETTEXTWINDOW**
GETCOLOR	**OUTTEXT**
GETCOLORRGB	**WRAPON**
GETTEXTCOLOR	

The following table lists other routines with the error or warning messages they produce for **GRSTATUS**:

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
ARC, ARC_W	$GRINVALIDPARAMETER	$GRNOOUTPUT
CLEARSCREEN	$GRINVALIDPARAMETER	
ELLIPSE, ELLIPSE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
FLOODFILLRGB	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
GETARCINFO	$GRERROR	
GETFILLMASK	$GRERROR, $GRINVALIDPARAMETER	
GETFONTINFO	$GRERROR	
GETGTEXTEXTENT	$GRERROR	
GETIMAGE	$GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
GETPIXEL	$GRBITMAPTOOLARGE	
GETPIXELRGB	$GRBITMAPTOOLARGE	
LINETO, LINETO_W		$GRNOOUTPUT, $GRCLIPPED
LOADIMAGE	$GRFILEOPENERROR, $GRNOBITMAPFILE, $GRALEREADERROR, $GRIMPROPERBITMAPFORMAT, $GRBITMAPTOOLARGE, $GRIMAGEREADERROR	
OUTGTEXT		$GRNOOUTPUT, $GRCLIPPED
PIE, PIE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
POLYGON, POLYGON_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED

PUTIMAGE, PUTIMAGE_W	$GRERROR, $GRINVALIDPARAMETER, $GRINVALIDIMAGEBUFFER $GRBITMAPDISPLAYERROR	$GRPARAMETERALTERED $GRNOOUTPUT
RECTANGLE, RECTANGLE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED
REMAPPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
REMAPALLPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
SAVEIMAGE	$GRFILEOPENERROR	
SCROLLTEXTWINDOW		$GRNOOUTPUT
SETBKCOLOR	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETBKCOLORRGB	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETCLIPRGN		$GRPARAMETERALTERED
SETCOLOR		$GRPARAMETERALTERED
SETCOLORRGB		$GRPARAMETERALTERED
SETFONT	$GRERROR, $GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
SETPIXEL, SETPIXEL_W		$GRNOOUTPUT
SETPIXELRGB, SETPIXELRGB_W		$GRNOOUTPUT
SETTEXTCOLOR		$GRPARAMETERALTERED
SETTEXTCOLORRGB		$GRPARAMETERALTERED
SETTEXTPOSITION		$GRPARAMETERALTERED
SETTEXTWINDOW		$GRPARAMETERALTERED
SETVIEWPORT		$GRPARAMETERALTERED

SETWINDOW	**$GRINVALIDPARAMETER**	**$GRPARAMETERALTERED**
SETWRITEMODE	**$GRINVALIDPARAMETER**	

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: ARC, ELLIPSE, FLOODFILLRGB, LINETO, PIE, POLYGON, REMAPALLPALETTERGB, SETBKCOLORRGB, SETCOLORRGB, SETPIXELRGB, SETTEXTCOLORRGB, SETWINDOW, SETWRITEMODE

HOSTNAM

Portability Function: Retrieves the current host computer name.

Module: USE DFPORT

Syntax

result = **HOSTNAM** (*name*)

name
(Output) Character*(*). Name of the current host. Should be at least as long as MAX_COMPUTERNAME_LENGTH, which is defined in the DFPORT module.

Results:

The result type is INTEGER(4). The result is zero if successful. If *name* is not long enough to contain all of the host name, the function truncates the host name and returns - 1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use dfport
character*20 hostnam
integer(4) istat
ISTAT = HOSTNAM (hostname)
```

HUGE

Elemental Intrinsic Function (Generic): Returns the largest number in the model representing the same type and kind parameters as the argument.

Syntax

result = **HUGE** (*x*)

x
(Input) Must be of type integer or real; it can be scalar or array valued.

Results:

The result type is 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 Model for Integer Data; real parameters *b*, *p*, and e_{max} are defined in Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: TINY, Data Representation Models

Examples

If X is of type REAL(4), HUGE (X) has the value $(1 - 2^{-24}) \times 2^{128}$.

IACHAR

Elemental Intrinsic Function (Generic): Returns the position of a character in the ASCII collating sequence.

Syntax

result = **IACHAR** (*c*)

c
(Input) Must be of type character of length 1.

Results:

The result type is 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ASCII and Key Code Charts, ACHAR, CHAR, ICHAR, LGE, LGT, LLE, LLT

Examples

IACHAR (’Y’) has the value 89.

IACHAR (’%’) has the value 37.

IAND

Elemental Intrinsic Function (Generic): Performs a logical AND on corresponding bits. This function can also be specified as **AND**.

Syntax

result = **IAND** (*i*, *j*)

i
(Input) Must be of type integer.

j
(Input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIAND	INTEGER(2)	INTEGER(2)
JIAND	INTEGER(4)	INTEGER(4)
KIAND [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IEOR, IOR, NOT

Examples

IAND (2, 3) has the value 2.

IAND (4, 6) has the value 4.

IARGC

Portability Function: Returns the index of the last command-line argument.

Module: USE DFPORT

Syntax

result = **IARGC** ( )

Results:

The result type is INTEGER(4). The result is the index of the last command-line argument.

IARGC is identical to the intrinsic function **NARGS**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NARGS

Example

```
use dfport
integer(4) no_of_arguments
no_of_arguments = IARGC ( )
print *, 'total command line arguments are ', no_of_arguments
```

IARGCOUNT (VMS only)

Inquiry Intrinsic Function (Generic): Returns the count of actual arguments passed to the current routine.

Syntax

result = **IARGCOUNT** ( )

Results:

The result type is default integer. Functions with a type of CHARACTER, COMPLEX(8), or REAL(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.

Examples

Consider the following:

```
CALL SUB (A,B)
...
SUBROUTINE SUB (X,Y,Z)
VOLATILE Z
TYPE *, IARGCOUNT()       ! Displays the value 2
```

IARGPTR

Inquiry Intrinsic Function (Generic): Returns a pointer to the actual argument list for the current routine.

Syntax

result = **IARGPTR** ( )

Results:

The result type is INTEGER(4) on Intel processors; INTEGER(8) on Alpha processors. The actual argument list is an array of values of the same type.

The first element has the address of the first argument. Formal (dummy) arguments that can be omitted must be declared **VOLATILE**.

See Also: VOLATILE

Example

```
WRITE (*, '(" Address of argument list is ", Z16.8)') IARGPTR ( )
```

IBCHNG

Elemental Intrinsic Function (Generic): Reverses the value of a specified bit in an integer.

Syntax

result = **IBCHNG** (*i*, *pos*)

i
(Input) Must be of type integer. This argument contains the bit to be reversed.

pos
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IAND, IBCLR, IBSET, IEOR, IOR, ISHA, ISHC, ISHL, ISHFT, NOT

Example

```
INTEGER J, K
J = IBCHNG(10, 2)     ! returns 14 = 1110
K = IBCHNG(10, 1)     ! returns 8 = 1000
```

IBCLR

Elemental Intrinsic Function (Generic): Clears one bit to zero.

Syntax

result = **IBCLR** (*i*, *pos*)

i
(Input) 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*).

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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBCLR	INTEGER(2)	INTEGER(2)
JIBCLR	INTEGER(4)	INTEGER(4)
KIBCLR [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IAND, IBCHNG, IBSET, IEOR, IOR, ISHA, ISHC, ISHL, ISHFT, NOT

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).

The following shows another example:

```
INTEGER J, K
J = IBCLR(7, 1) ! returns 5 = 0101
K = IBCLR(5, 1) ! returns 5 = 0101
```

IBITS

Elemental Intrinsic Function (Generic): Extracts a sequence of bits (a bit field).

Syntax

result = **IBITS** (*i*, *pos*, *len*)

i
(Input) Must be of type integer.

pos
(Input) Must be of type integer. It must not be negative and *pos* + *len* must be less than or equal to **BIT_SIZE**(*i*).

len
(Input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBITS	INTEGER(2)	INTEGER(2)
JIBITS	INTEGER(4)	INTEGER(4)

KIBITS [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, BIT_SIZE, IBCLR, IBSET, ISHFT, ISHFTC, MVBITS

Examples

IBITS (12, 1, 4) has the value 6.

IBITS (10, 1, 7) has the value 5.

IBSET

Elemental Intrinsic Function (Generic): Sets one bit to 1.

Syntax

result = **IBSET** (*i*, *pos*)

i
(Input) Must be of type integer.

pos
(Input) Must be of type integer. It must not be negative and it must be less than **BIT_SIZE**(*i*).

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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIBSET	INTEGER(2)	INTEGER(2)
JIBSET	INTEGER(4)	INTEGER(4)
KIBSET [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IAND, IBCHNG, IBCLR, IEOR, IOR, ISHA, ISHC, ISHL, ISHFT, NOT

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).

The following shows another example:

```
INTEGER I
I = IBSET(8, 2) ! returns 12 = 1100
```

ICHAR

Elemental Intrinsic Function (Generic): Returns the position of a character in the ASCII character set.

Syntax

result **= ICHAR** (*c*)

c
(Input) Must be of type character of length 1.

Results:

The result type is default integer. The result value is the position of *c* in the ASCII 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) [2]

[1] This specific function cannot be passed as an actual argument.
[2] INTEGER(8) is only available on Alpha processors.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IACHAR, CHAR, ASCII and Key Code Charts

Examples

ICHAR ('W') has the value 87.

ICHAR ('#') has the value 35.

IDATE

IDATE can be used as an intrinsic subroutine or as a portability routine.

> **Warning:** The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

IDATE Intrinsic Subroutine

Intrinsic Subroutine: Returns three integer values representing the current month, day, and year.

Syntax

> **CALL IDATE** (*i*, *j*, *k*)
>
> *i*
> Is the current month.
>
> *j*
> Is the current day.
>
> *k*
> Is the current year.

Example

If the current date is September 16, 1996, the values of the integer variables upon return are: I = 9, J = 16, and K = 96.

IDATE Portability Routine

Portability Subroutine: Returns the month, day, and year of the current system.

Module: USE DFPORT

Syntax

CALL IDATE (*i*, *j*, *k*)

-or-

CALL IDATE (*iarray*)

i
(Output) INTEGER(4). Current system month.

j
(Output) INTEGER(4). Current system day.

k
(Output) INTEGER(4). Current system year as an offset from 1900.

iarray
(Output) INTEGER(4). Three-element array that holds day as element 1, month as element 2, and year as element 3. The month is between 1 and 12 and the year is greater than or equal to 1969.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE, DATE_AND_TIME, GETDAT

Example

```
      use dfport
      integer(4) imonth, iday, iyear, datarray(3)
! If the date is July 11, 1996:
    CALL IDATE(IMONTH, IDAY, IYEAR)
! sets IMONTH to 7, IDAY to 11 and IYEAR to 96.
    CALL IDATE (DATARRAY)
! datarray is (/11,7,96/)
```

IDENT

Compiler 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.

Syntax

*c***DEC$ IDENT** *string*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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.

See Also: General Compiler Directives

IEOR

Elemental Intrinsic Function (Generic): Performs an exclusive OR on corresponding bits. This function can also be specified as **XOR**.

Syntax

result = **IEOR** (*i*, *j*)

i
(Input) Must be of type integer.

j
(Input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIEOR	INTEGER(2)	INTEGER(2)
JIEOR	INTEGER(4)	INTEGER(4)

KIEOR [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IAND, IOR, NOT

Examples

IEOR (1, 4) has the value 5.

IEOR (3, 10) has the value 9.

The following shows another example:

```
INTEGER I
I = IEOR(240, 90)    ! returns 170
```

IERRNO

Portability Function: Returns the number of the last detected error from any routines in the DFPORT module that return error codes.

Module: USE DFPORT

Syntax

result = **IERRNO** ( )

Results:

Type INTEGER(4). Last error code from any portability routines that return error codes. These error codes are analogous to *errno* on a U*X system. The module DFPORT.F90 (in \DF98\INCLUDE) provides parameter definitions for the following Unix *errno* names (typically found in errno.h on U*X systems).

Symbolic name	Number	Description
EPERM	1	Insufficient permission for operation
ENOENT	2	No such file or directory
ESRCH	3	No such process
EIO	5	I/O error

E2BIG	7	Argument list too long
ENOEXEC	8	File is not executable
ENOMEM	12	Not enough resources
EACCES	13	Permission denied
EXDEV	18	Cross-device link
ENOTDIR	20	Not a directory
EINVAL	22	Invalid argument

The value returned by **IERRNO** is updated only when an error occurs. For example, if an error occurs on a **GETLOG** call and then two **CHMOD** calls succeed, a subsequent call to **IERRNO** returns the error for the **GETLOG** call.

Examine **IERRNO** immediately after returning from a Portability routine. Other Fortran routines, as well as any Win32 APIs, can also change the error code to an undefined value. **IERRNO** is set on a per thread basis.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFPORT
CHARACTER*20 username
INTEGER(4) ierrval
ierrval=0 !initialize return value
CALL GETLOG(username)
IF (IERRNO( ) == ierrval) then
  print *, 'User name is ',username
  exit
ELSE
  ierrval = ierrno()
  print *, 'Error is ',ierrval
END IF
```

IF -- Arithmetic

Statement: Conditionally transfers control to one of three statements, based on the value of an arithmetic expression. (It is an obsolescent feature in Fortran 90 and Fortran 95.)

Syntax

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*

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SELECT CASE ... END SELECT, Execution Control, Obsolescent Features in Fortran 90

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
```

The following statement transfers control to statement 10 for n < 10, to statement 20 for n = 10, and to statement 30 for n > 10:

```
IF (n-10) 10, 20, 30
```

The following statement transfers control to statement 10 if n<= 10, and to statement 30 for n>10:

```
IF (n-10) 10, 10, 30
```

IF -- Logical

Statement: Conditionally executes one statement based on the value of a logical expression. (This statement was called a logical IF statement in FORTRAN 77.)

Syntax

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**, or **IF** 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IF Construct, Execution Control

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
```

The following shows another example:

```
   USE DFPORT
   INTEGER(4) istat, errget
   character(inchar)
   real x
   istat = getc(inchar)
   IF (istat) errget = -1
   ...
!
```

```
IF (x .GT. 2.3) call new_subr(x)
...
```

IF Construct

Statement: 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.)

Syntax

[*name*:] **IF** (*expr*) **THEN**
    block
[**ELSE IF** (*expr*) **THEN** [*name*]
    block]...
[**ELSE** [*name*]
    block]
END IF [*name*]

name
(Optional) 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
```

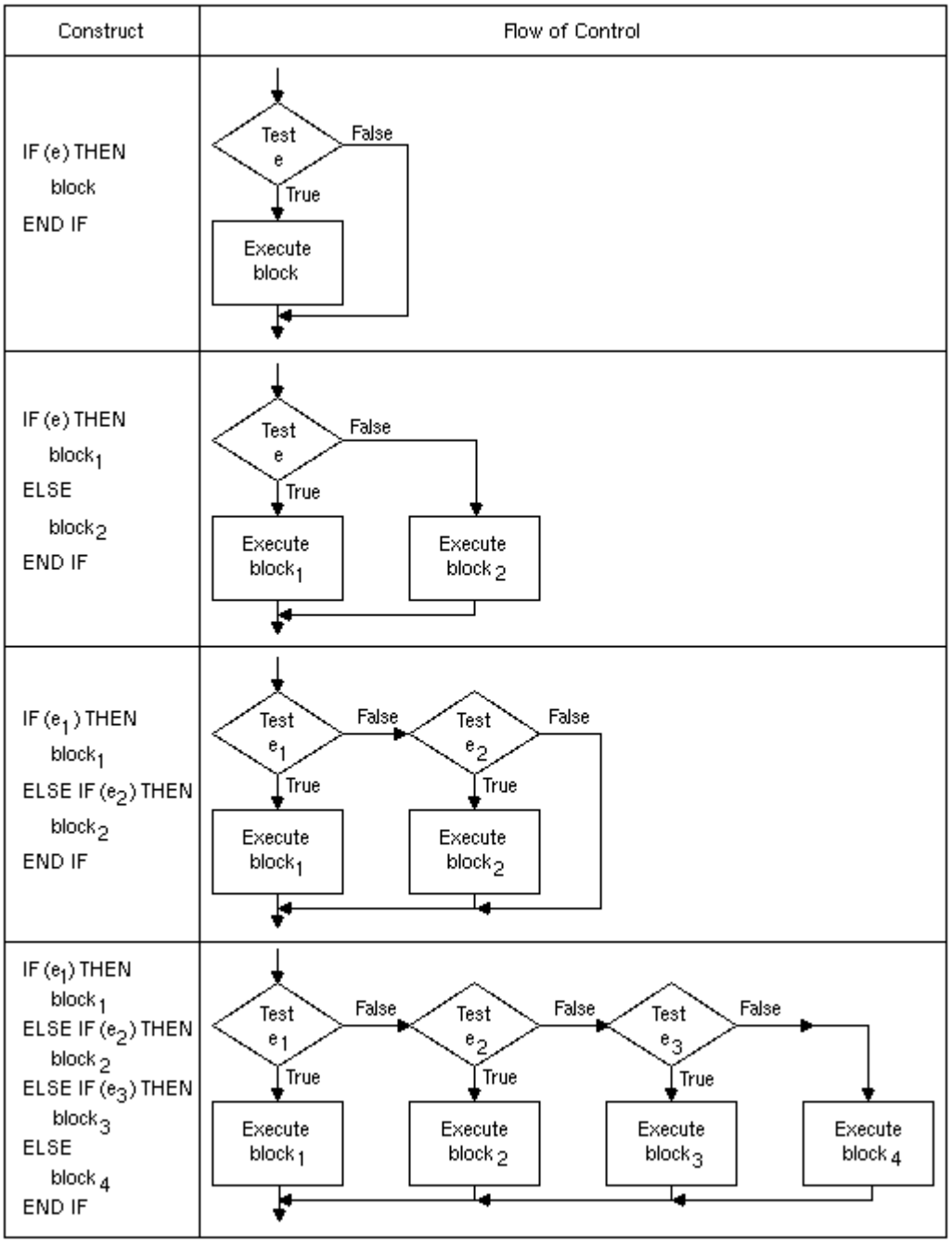
This statement is translated as the following logical **IF** statement:

```
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.

The following figure shows the flow of control in **IF** constructs:

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Execution Control, IF -- Logical, IF -- Arithmetic

Examples

Examples

The following example shows the simplest form of an **IF** construct:

```
Form                  Example
IF (expr) THEN        IF (ABS(ADJU) .GE. 1.0E-6) THEN
  block                 TOTERR = TOTERR + ABS(ADJU)
                        QUEST = ADJU/FNDVAL
END IF                END IF
```

This construct conditionally executes the block of statements between the **IF THEN** and the **END IF** statements.

The following shows another example:

```
!  Simple block IF:
   IF (i .LT. 10) THEN
      ! the next two statements are only executed if i < 10
      j = i
      slice = TAN (angle)
   END IF
```

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
```

The following example shows an **IF** construct containing an **ELSE** statement:

```
Form                  Example
IF (expr) THEN        IF (NAME .LT. 'N') THEN
  block1                IFRONT = IFRONT + 1
                        FRLET(IFRONT) = NAME(1:2)
ELSE                  ELSE
  block2                IBACK = IBACK + 1
END IF                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          IF (A .GT. B) THEN
  block1                  D = B
                          F   A   B
```

```
                          F = A - B
ELSE IF (expr) THEN       ELSE IF (A .GT. B/2.) THEN
  block2                    D = B/2.
                            F = A - B/2.
END IF                    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 shows another example:

```
!  Block IF with ELSE IF statements:

   IF (j .GT. 1000) THEN
      ! Statements here are executed only if J > 1000
   ELSE IF (j .GT. 100) THEN
      ! Statements here are executed only if J > 100 and j <= 1000
   ELSE IF (j .GT. 10) THEN
      ! Statements here are executed only if J > 10 and j <= 100
   ELSE
      ! Statements here are executed only if j <= 10
   END IF
```

The following example shows an IF construct containing several **ELSE IF THEN** statements and an **ELSE** statement:

```
  Form                    Example
  IF (expr) THEN          IF (A .GT. B) THEN
    block1                  D = B
                            F = A - B
  ELSE IF (expr) THEN     ELSE IF (A .GT. C) THEN
    block2                  D = C
                            F = A - C
  ELSE IF (expr) THEN     ELSE IF (A .GT. Z) THEN
    block3                  D = Z
                            F = A - Z
  ELSE                    ELSE
    block4                  D = 0.0
                            F = A
  END IF                  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 (expr) THEN          IF (A .LT. 100) THEN
    block1                  INRAN = INRAN + 1
    IF (expr2) THEN         IF (ABS(A-AVG) .LE. 5.) THEN
      block1a                 INAVG = INAVG + 1
    ELSE                    ELSE
      block1b                 OUTAVG = OUTAVG + 1
```

```
      block1b                    OUTAVG   OUTAVG + 1
    END IF                     END IF
  ELSE                       ELSE
    block2                     OUTRAN = OUTRAN + 1
  END IF                     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 shows another example:

```
!  Nesting of constructs and use of an ELSE statement following
!  a block IF without intervening ELSE IF statements:

   IF (i .LT. 100) THEN
      ! Statements here executed only if i < 100
      IF (j .LT. 10) THEN
      ! Statements here executed only if i < 100 and j < 10
      END IF
      ! Statements here executed only if i < 100
   ELSE
      ! Statements here executed only if i >= 100
      IF (j .LT. 10) THEN
      ! Statements here executed only if i >= 100 and j < 10
      END IF
      ! Statements here executed only if i >= 100
   END IF
```

IF Directive Construct

Compiler Directive: A conditional compilation construct that begins with an **IF** or **IF DEFINED** directive. **IF** tests whether a logical expression is .TRUE. or .FALSE.. **IF DEFINED** tests whether a symbol has been defined.

Syntax

*c***DEC$ IF** (*expr*) -or- *c***DEC$ IF DEFINED** (*name*)
block
[*c***DEC$ ELSEIF** (*expr*)
block]...
[*c***DEC$ ELSE**
block]
*c***DEC$ ENDIF**

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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.

Each directive in the construct can begin with `!MS$` instead of `cDEC$`.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DEFINE and UNDEFINE, IF Construct, General Compiler Directives

Example

```
!  When the following code is compiled and run,
!  the output is:
!  Or this compiled if all preceding conditions .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
```

IF DEFINED Directive

See the IF Directive Construct.

IFIX

Elemental Intrinsic Function (Generic): Converts a single-precision real argument to an integer by truncating. For more information, see INT.

ILEN

Elemental Function (Generic): Returns the length (in bits) of the two's complement representation of an integer.

Syntax

result = **ILEN** (*i*)

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.
```

IMAGESIZE, IMAGESIZE_W

Graphics Function: Returns the number of bytes needed to store the image inside the specified

bounding rectangle. **IMAGESIZE** is useful for determining how much memory is needed for a call to **GETIMAGE**.

Module: USE DFLIB

Syntax

result = **IMAGESIZE** (*x1*, *y1*, *x2*, *y2*)
result = **IMAGESIZE_W** (*wx1*, *wy1*, *wx2*, *wy2*)

x1, *y1*
(Input) INTEGER(2). Viewport coordinates for upper-left corner of image.

x2, *y2*
(Input) INTEGER(2). Viewport coordinates for lower-right corner of image.

wx1, *wy1*
(Input) REAL(8). Window coordinates for upper-left corner of image.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of image.

Results:

The result type is INTEGER(4). The result is the storage size of an image in bytes.

IMAGESIZE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). **IMAGESIZE_W** defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

IMAGESIZE_W defines the bounding rectangle in terms of window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETIMAGE, GRSTATUS, PUTIMAGE

Example

See the example in GETIMAGE.

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:

Syntax

IMPLICIT *type* (*a*[, *a*]...)[, *type* (*a*[, *a*]...)]...
IMPLICIT NONE

type
Is a data type specifier (CHARACTER*(*) is not allowed).

a
Is a single letter 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).

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 2**31-1 for DIGITAL UNIX and Windows NT systems on Alpha processors; 1 to 65535 for OpenVMS systems and Intel processors.

Names beginning with a dollar sign ($) are implicitly INTEGER; the data type cannot be changed in an **IMPLICIT** statement.

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 the /warn:declarations complier 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)
```

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Data Types, Constants, and Variables

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)
```

The following shows another example:

```
IMPLICIT INTEGER (a-b), CHARACTER*10 (n), TYPE(fried) (c-d)

TYPE fried
  INTEGER e, f
  REAL g, h
END TYPE
age = 10       ! integer
name = 'Paul'  ! character
c%e = 1        ! type fried, integer component
```

INCHARQQ

QuickWin Function: Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering.

Module: USE DFLIB

Syntax

result = **INCHARQQ** ( )

Results:

The result type is INTEGER(2). The result is the ASCII key code.

The keystroke is read from the child window that currently has the focus. You must call **INCHARQQ** before the keystroke is made (**INCHARQQ** does not read the keyboard buffer). This function does not echo its input. For function keys, **INCHARQQ** returns 0xE0 as the upper 8 bits, and the ASCII code as the lower 8 bits.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, GETCHARQQ, READ, MBINCHARQQ, GETC.

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:

Syntax

INCLUDE *'filename* [/[NO]LIST]'

INCLUDE '[*text-lib*] (*module-name*) [/[NO]LIST]' (VMS only)

filename
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.

On Windows NT and Windows 95 systems, you can only use /[NO]LIST if you specify the /vms compiler option (which sets OpenVMS defaults).

text-lib (VMS only)
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 (VMS only)
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 be up to 31 characters long and can contain any alphanumeric character and the special characters dollar sign ($) and underscore (_).

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

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 Statements.

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.

You can use modules instead of include files to achieve encapsulation of related data types and procedures. For example, one module can contain derived type definitions as well as special operators and procedures that apply to those types. For information on how to use modules, see Program Units and Procedures.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MODULE, USE

Examples

In the following example, a file named COMMON.FOR (in the current working directory) is included and read as input.

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
  RETURN
END
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

The file COMMON.FOR defines a named constant M, and defines arrays X and Y as part of blank common.

The following example program declares its common data in an include file. The contents of the file INCLUDE.INC are inserted in the source code in place of every **INCLUDE** 'INCLUDE.INC' statement. This guarantees that all references to common storage variables are consistent.

```
INTEGER i
REAL x
INCLUDE 'INCLUDE.INC'

DO i = 1, 5
  READ (*, '(F10.5)') x
  CALL Push (x)
END DO
```

INDEX

Elemental Intrinsic Function (Generic): Returns the starting position of a substring within a string.

Syntax

result = **INDEX** (*string*, *substring* [, *back*])

string
(Input) Must be of type character.

substring
(Input) Must be of type character.

back
(Optional; input) Must be of type logical.

Results:

The result type is default integer.

If *back* does not appear (or appears with the value 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* appears with the value 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)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SCAN

Examples

INDEX (’FORTRAN’, ’O’, BACK = .TRUE.) has the value 2.

INDEX (’XXXX’, " ", BACK = .TRUE.) has the value 5.

The following shows another example:

```
I = INDEX(’banana’,’an’, BACK = .TRUE.) ! returns 4
I = INDEX(’banana’, ’an’) ! returns 2
```

INITIALIZEFONTS

QuickWin Function: Initializes Windows fonts.

Module: USE DFLIB

Syntax

result = **INITIALIZEFONTS** ( )

Results:

The result type is INTEGER(2). The result is the number of fonts initialized.

All fonts in Windows become available after a call to **INITIALIZEFONTS**. Fonts must be initialized with **INITIALIZEFONTS** before any other font-related library function (such as **GETFONTINFO**, **GETGTEXTEXTENT**, **SETFONT**, **OUTGTEXT**) can be used. For more information, see Using Fonts from the Graphics Library in the *Programmer’s Guide*.

The font functions affect the output of **OUTGTEXT** only. They do not affect other Fortran I/O functions (such as **WRITE**) or graphics output functions (such as **OUTTEXT**).

For each window you open, you must call **INITIALIZEFONTS** before calling **SETFONT**. **INITIALIZEFONTS** needs to be executed after each new child window is opened in order for a subsequent **SETFONT** call to be successful.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, SETFONT, OUTGTEXT.

Example

```
! build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(2) numfonts
numfonts = INITIALIZEFONTS()
WRITE (*,*) numfonts
END
```

INITIALSETTINGS

QuickWin Function: Initializes QuickWin.

Module: USE DFLIB

Syntax

result = **INITIALSETTINGS** ( )

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can change the initial appearance of an application's default frame window and menus by defining an **INITIALSETTINGS** function. If no user-defined **INITIALSETTINGS** function is supplied, QuickWin calls a predefined **INITIALSETTINGS** routine to control the default frame window and menu appearance. You do not need to call **INITIALSETTINGS** unless you define it. For more information, see Controlling the Initial Menu and Frame Window in the *Programmer's Guide*.

Compatibility

QUICKWIN GRAPHICS WINDOWS LIB

See Also: Using QuickWin, APPENDMENUQQ, INSERTMENUQQ, DELETEMENUQQ.

INQFOCUSQQ

QuickWin Function: Determines which window has the focus.

Module: USE DFLIB

Syntax

result = **INQFOCUSQQ** (*unit*)

unit
(Output) INTEGER(4). Unit number of the window that has the I/O focus.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero. The function fails if the window with the focus is associated with a closed unit.

Unit numbers 0, 5, and 6 refer to the default window only if the program has not specifically opened them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

The window with focus is always in the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling **SETACTIVEQQ**.

A window has focus when it is given the focus by **FOCUSQQ**, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

IOFOCUS defaults to .TRUE., except for child windows opened as unit *, in which case IOFOCUS defaults to .FALSE.. If IOFOCUS=.TRUE., the child window receives focus prior to each **READ**, **WRITE**, **PRINT**, or **OUTTEXT**. Calls to graphics functions (such as **OUTGTEXT** and **ARC**) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: FOCUSQQ, *Programmer's Guide*: Using QuickWin

INQUIRE

Statement: Returns information on the status of specified properties of a file or logical unit. It takes one of the following forms:

Syntax

Inquiring by File:

INQUIRE (FILE=*name* [, ERR=*label*] [, IOSTAT=*i-var*] [, DEFAULTFILE=*def*], *slist*)

Inquiring by Unit:

INQUIRE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*] *slist*)

Inquiring by Output List:

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.

slist
Is one or more of the following inquiry specifiers (each specifier can appear only once):

ACCESS	DELIM	NAMED	READWRITE
ACTION	DIRECT	NEXTREC	RECL
BINARY	EXIST	NUMBER	RECORDTYPE
BLANK	FORM	OPENED	SEQUENTIAL
BLOCKSIZE	FORMATTED	ORGANIZATION	SHARE
BUFFERED	IOFOCUS	PAD	UNFORMATTED
CARRIAGECONTROL	MODE	POSITION	WRITE
CONVERT	NAME	READ	

def
Is the label of the branch target statement that receives control if an error occurs. Is a scalar default character expression specifying a default file pathname string. (For more information, see the DEFAULTFILE specifier under **OPEN** in the *Language Reference*.

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
(Output) Is a scalar default integer variable indicating the number of bytes of data that would result from using *out-item-list* in an unformatted output statement.

out-item-list
(Output) Is a list of one or more output items (see I/O Lists.

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 keyword 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.

To inquire about a file using the DEFAULTFILE specifier, the specifier must also appear in the **OPEN** statement for that file. You can specify DEFAULTFILE=*def* in addition to (or in place of) FILE=*name*, and the *name* and *def* can start with a tilde (~).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INQUIRE Statement, OPEN

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.

The following shows another example:

```
!    This program prompts for the name of a data file.
!    The INQUIRE statement then determines whether
!    the file exists. If it does not, the program
!    prompts for another file name.

      CHARACTER*12 fname
      LOGICAL exists

!     Get the name of a file:
100   WRITE (*, '(1X, A\)') 'Enter the file name: '
      READ (*, '(A)') fname

      INQUIRE about file's existence:
```

```
!     INQUIRE about file's existence:
      INQUIRE (FILE = fname, EXIST = exists)

      IF (.NOT. exists) THEN
        WRITE (*,'(2A/)') ' >> Cannot find file ', fname
        GOTO 100
      END IF
      END
```

INSERTMENUQQ

QuickWin Function: Inserts a menu item into a QuickWin menu and registers its callback routine.

Module: USE DFLIB

Syntax

result = **INSERTMENUQQ** (*menuID, itemID, flag, text, routine*)

menuID
(Input) INTEGER(4). Identifies the menu in which the item is inserted, starting with 1 as the leftmost menu.

itemID
(Input) INTEGER(4). Identifies the position in the menu where the item is inserted, starting with 0 as the top menu item.

flag
(Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see Results section below). The following constants are available:

- **$MENUGRAYED**: Disables and grays out the menu item.
- **$MENUDISABLED**: Disables but does not gray out the menu item.
- **$MENUENABLED**: Enables the menu item.
- **$MENUSEPARATOR**: Draws a separator bar.
- **$MENUCHECKED**: Puts a check by the menu item.
- **$MENUUNCHECKED**: Removes the check by the menu item.

text
(Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, words of text'C.

routine
(Input) EXERNAL. Callback subroutine that is called if the menu item is selected. All routines must take a single LOGICAL parameter which indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- **WINPRINT**: Prints the program.
- **WINSAVE**: Saves the program.
- **WINEXIT**: Terminates the program.
- **WINSELTEXT**: Selects text from the current window

- **WINSELTEXT**: Selects text from the current window.
- **WINSELGRAPH**: Selects graphics from the current window.
- **WINSELALL**: Selects the entire contents of the current window.
- **WINCOPY**: Copies the selected text and/or graphics from current window to the Clipboard.
- **WINPASTE**: Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a **READ.**
- **WINCLEARPASTE**: Clears the paste buffer.
- **WINSIZETOFIT**: Sizes output to fit window.
- **WINFULLSCREEN**: Displays output in full screen.
- **WINSTATE**: Toggles between pause and resume states of text output.
- **WINCASCADE**: Cascades active windows.
- **WINTILE**: Tiles active windows.
- **WINARRANGE**: Arranges icons.
- **WINSTATUS**: Enables a status bar.
- **WININDEX**: Displays the index for QuickWin help.
- **WINUSING**: Displays information on how to use Help.
- **WINABOUT**: Displays information about the current QuickWin application.
- **NUL**: No callback routine.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE.

Menus and menu items must be defined in order from left to right and top to bottom. For example, **INSERTMENUQQ** fails if you try to insert menu item 7 when 5 and 6 are not defined yet. For a top-level menu item, the callback routine is ignored if there are subitems under it.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to **INSERTMENUQQ** as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

For more information on customizing QuickWin menus, see Using QuickWin in the *Programmer's Guide*.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: APPENDMENUQQ, DELETEMENUQQ, MODIFYMENUFLAGSQQ, MODIFYMENUROUTINEQQ, MODIFYMENUSTRINGQQ.

Example

```
! build as a QuickWin App.
USE DFLIB
LOGICAL(4) status
! insert new item into Menu 5 (Window)
status= INSERTMENUQQ(5, 5, $MENUCHECKED, 'New Item'C, &
                     WINSTATUS)
! insert new menu in position 2
status= INSERTMENUQQ(2, 0, $MENUENABLED, 'New Menu'C, &
                     WINSAVE)
END
```

INT

Elemental Intrinsic Function (Generic): Converts a value to integer type.

Syntax

result = **INT** (*a* [, *kind*])

a
(Input) Must be of type integer, real, or complex.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is default integer. (If the processor cannot represent the result in integer type, the result is undefined.) 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.

The result value depends on the type and absolute value of *a*, as follows:

- 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*.

Specific Name [1]	**Argument Type** [2]	**Result Type** [2]
	INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)

	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8)	INTEGER (8)
IIFIX [3]	REAL(4)	INTEGER (2)
IINT	REAL(4)	INTEGER (2)
IFIX [4, 5]	REAL(4)	INTEGER (4)
JFIX	INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), COMPLEX(4), COMPLEX(8)	INTEGER (4)
INT [6, 7]	REAL(4)	INTEGER (4)
KIFIX	REAL(4)	INTEGER (8)
KINT	REAL(4)	INTEGER (8)
IIDINT	REAL(8)	INTEGER (2)
IDINT [7, 8]	REAL(8)	INTEGER (4)
KIDINT	REAL(8)	INTEGER (8)
IIQINT	REAL(16)	INTEGER (2)
IQINT [7, 9]	REAL(16)	INTEGER (4)
KIQINT	REAL(16)	INTEGER (8)
	COMPLEX(4), COMPLEX(8)	INTEGER (2)
	COMPLEX(4), COMPLEX(8)	INTEGER (4)
	COMPLEX(4), COMPLEX(8)	INTEGER (8)

INT1	INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), COMPLEX(4), COMPLEX(8)	INTEGER (1)
INT2	INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), COMPLEX(4), COMPLEX(8)	INTEGER (2)
INT4	INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), COMPLEX(4), COMPLEX(8)	INTEGER (4)

[1] These specific functions cannot be passed as actual arguments.
[2] INTEGER(8) is only available on Alpha processors; REAL(16) is available on OpenVMS and DIGITAL UNIX systems.
[3] This function can also be specified as HFIX.
[4] The setting of compiler option /integer_size or /real_size can affect IFIX.
[5] For compatibility with older versions of Fortran, IFIX can also be specified as a generic function.
[6] Or JINT.
[7] The setting of compiler option /integer_size can affect INT, IDINT, and IQINT.
[8] Or JIDINT. For compatibility with older versions of Fortran, IDINT can also be specified as a generic function.
[9] Or JIQINT. For compatibility with older versions of Fortran, IQINT can also be specified as a generic function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NINT, AINT, ANINT, REAL, DBLE, SNGL

Examples

INT (-4.2) has the value -4.

INT (7.8) has the value 7.

INTEGER Directive

Compiler Directive: Specifies the default integer kind.

Syntax

*c***DEC$ INTEGER**:{ 2 | 4 | 8 }

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

Rules and Behavior

The **INTEGER** directive specifies a size of 2 (KIND=2), 4 (KIND=4), or 8 (KIND=8) bytes for default integer numbers. INTEGER(KIND=8) is only available on Alpha processors.

When the **INTEGER** directive is in effect, all default integer variables are of the kind specified. 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. **INTEGER** 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.

The following form is also allowed: `!MS$INTEGER:{2|4|8}`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INTEGER, REAL Directive, General Compiler Directives.

Example

```
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
```

INTEGER

Statement: Specifies the INTEGER data type.

Syntax

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

n
Is kind 1, 2, 4, or 8. Kind 8 is only available on Alpha processors.

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 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.

The default kind can also be changed by using the INTEGER directive or the /integer_size compiler option.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INTEGER Directive, *Programmer's Guide*: Integer Data Types and Integer Constants.

Examples

```
! Entity-oriented declarations:
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER (2) :: k=4
INTEGER (2), PARAMETER :: limit=12

! Attribute-oriented declarations:
INTEGER days, hours
INTEGER (2):: k=4, limit
DIMENSION days(:), hours(:)
POINTER days, hours
PARAMETER (limit=12)
```

INTEGERTORGB

QuickWin Subroutine: Converts an RGB color value into its red, green, and blue components.

Module: USE DFLIB

Syntax

CALL INTEGERTORGB (*rgb*, *red*, *green*, *blue*)

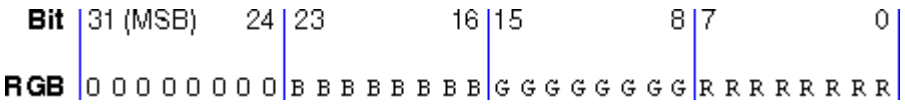
rgb
(Input) INTEGER(4). RGB color value whose red, green, and blue components are to be returned.

red
(Output) INTEGER(4). Intensity of the red component of the RGB color value.

green
(Output) INTEGER(4). Intensity of the green component of the RGB color value.

blue
(Output) INTEGER(4). Intensity of the blue component of the RGB color value.

INTEGERTORGB separates the four-byte RGB color value into the three components as follows:



Compatibility

QUICKWIN GRAPHICS WINDOWS LIB

See Also: Using QuickWin, RGBTOINTEGER, GETCOLORRGB, GETBKCOLORRGB, GETPIXELRRGB, GETPIXELSRGB, GETTEXTCOLORRGB.

Example

```
! build as a QuickWin App.
USE DFLIB
INTEGER(4) r, g, b

CALL INTEGERTORGB(2456, r, g, b)
write(*,*) r, g, b
END
```

INTENT

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **INTENT** (*intent-spec*) [, *att-ls*] **::** *d-arg* [, *d-arg*]...

Statement:

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

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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: Argument Association, Type Declarations, Compatible attributes.

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
   ...
```

The following shows another example:

```
!Calculate value into a running average and return the average cubed.

TYPE DATA
  INTEGER count
  REAL average
END TYPE
. . .
SUBROUTINE AVERAGE(value,data1,cube_ave)
  TYPE(DATA) data1
  REAL dummy
  ! value cannot be changed, while cube_ave must be defined
  ! before it can be used. Data1 is defined when the procedure is
  ! invoked, and becomes redefined in the subroutine.
  INTENT(IN)::value; INTENT(OUT)::cube_ave
  INTENT(INOUT)::data1

  ! count number of times AVERAGE has been called on the data set
  ! being passed.
  dummy = count*average + value
  data1%count = data1%count + 1
  data1%average = dummy/data1%count
  cube_ave = data1%average**3
END SUBROUTINE
```

INTERFACE

Statement: Defines 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.

Syntax

INTERFACE [*generic-spec*]
[*interface-body*] ...
[**MODULE PROCEDURE** *name-list*] ...
END INTERFACE [*generic-spec*]

generic-spec
(Optional) Is one of the following:

- A generic name
  For information on generic names, see Program Units and Procedures.

- **OPERATOR** (*op*)

  Defines a generic operator (*op*). It can be a defined unary, defined binary, or extended intrinsic operator. For information on defined operators, see Program Units and Procedures.

- **ASSIGNMENT** (=)

  Defines generic assignment. For information on defined assignment, see Assignment -- Defined 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 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.

Internal, module, and intrinsic procedures are all considered to have explicit interfaces. External procedures have implicit interfaces by default; when you specify an interface block for them, their interface becomes explicit. A procedure must not have more than one explicit interface in a given scoping unit. This means that you cannot include internal, module, or intrinsic procedures in an interface block, unless you want to define a generic name for them.

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 Unambiguous Generic Procedure References.

- 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CALL, FUNCTION, MODULE, MODULE PROCEDURE, SUBROUTINE, PURE, Procedure Interfaces, Use and Host Association, Determining When Procedures Require Explicit Interfaces, Defining Generic Names for Procedures, Defining Generic Operators, Defining Generic Assignment

Examples

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
```

The following shows another example:

```
!An interface to an external subroutine SUB1 with header:
!SUBROUTINE SUB1(I1,I2,R1,R2)
!INTEGER I1,I2
!REAL R1,R2

INTERFACE
  SUBROUTINE SUB1(int1,int2,real1,real2)
    INTEGER int1,int2
    REAL real1,real2
  END SUBROUTINE SUB1
END INTERFACE

INTEGER int
. . .
```

INTRINSIC

Statement and 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 Functions Not Allowed as Actual Arguments.)

The INTRINSIC attribute can be specified in a type declaration statement or an **INTRINSIC** statement, and takes one of the following forms:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **INTRINSIC** [, *att-ls*] **::** *in-pro* [, *in-pro*]...

Statement:

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: References to Generic Procedures, Type Declarations, Compatible attributes.

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`	`SUBROUTINE TRIG(X,F,Y)`
`INTRINSIC SIN, COS`	`Y = F(X)`
`...`	`RETURN`
	`END`
`CALL TRIG(ANGLE,SIN,SINE)`	
`...`	`FUNCTION CTN(X)`
	`CTN = COS(X)/SIN(X)`
`CALL TRIG(ANGLE,COS,COSINE)`	`RETURN`
`...`	`END`
`CALL TRIG(ANGLE,CTN,COTANGENT)`	

Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Fortran 90 library functions **SIN** and **COS**; but when TRIG is called with a second

argument of CTN, F(X) references the user function CTN.

The following shows another example:

```
      INTRINSIC SIN, COS
      REAL X, Y, R
!     SIN and COS are arguments to Calc2:
      R = Calc2 (SIN(x), COS(y))
```

IOR

Elemental Intrinsic Function (Generic): Performs an inclusive OR on corresponding bits. This function can also be specified as **OR**.

Syntax

result = **IOR** (*i*, *j*)

i
(Input) Must be of type integer.

j
(Input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IIOR	INTEGER(2)	INTEGER(2)
JIOR	INTEGER(4)	INTEGER(4)
KIOR [1]	INTEGER(8)	INTEGER(8)

[1] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: IAND, IEOR, NOT

Examples

IOR (1, 4) has the value 5.

IOR (1, 2) has the value 3.

The following shows another example:

```
INTEGER result
result = IOR(240, 90) ! returns 250
```

IRAND, IRANDM

Portability Functions: Return random numbers in the range 0 through (2**31)-1, or 0 through (2**15)-1 if called without an argument.

Module: USE DFPORT

Syntax

result = **IRAND** ([*iflag*])
result = **IRANDM** (*iflag*)

iflag
(Input) INTEGER(4). Optional for **IRAND**. Controls the way the returned random number is chosen. If *iflag* is omitted, it is assumed to be 0, and the return range is 0 through (2**15)-1 (inclusive).

Results:

The result type is INTEGER(4). If *iflag* is 1, the generator is restarted and the first random value is returned. If *iflag* is 0, the next random number in the sequence is returned. If *iflag* is neither zero nor 1, it is used as a new seed for the random number generator, and the functions return the first new random value.

IRAND and **IRANDM** are equivalent and return the same random numbers. Both functions are included to ensure portability of existing code that references one or both of them.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_NUMBER, RANDOM_SEED, Portability Library

Example

```
USE DFPORT
INTEGER(4) istat, flag_value, r_nums(20)
flag_value=1
r_nums(1) = IRAND (flag_value)
flag_value=0
do istat=2,20
   r_nums(istat) = irand(flag_value)
end do
```

ISHA

Elemental Function (Generic): Arithmetically shifts an integer left or right by a specified number of bits.

Syntax

result = **ISHA** (*i*, *shift*)

i
(Input) Must be of type integer. This argument is the value to be shifted.

shift
(Input) 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. Zeros are shifted in from the opposite end.

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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ISHC, ISHL, ISHFT, ISHFTC

Example

```
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 10100000 = -2048
```

ISHC

Elemental Intrinsic Function (Generic): 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.

Syntax

result = **ISHC** (*i*, *shift*)

i
(Input) Must be of type integer. This argument is the value to be rotated.

shift
(Input) 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ISHC, ISHA, ISHL, ISHFT

Example

```
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
```

ISHFT

Elemental Intrinsic Function (Generic): Performs a logical shift.

Syntax

result = **ISHFT** (*i*, *shift*)

i
(Input) Must be of type integer.

shift
(Input) 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, 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. Zeros are shifted in from the opposite end.

ISHFT with a positive *shift* can also be specified as **LSHIFT**. **ISHFT** with a negative *shift* can also be specified as **RSHIFT** with |*shift*|.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IISHFT	INTEGER(2)	INTEGER(2)
JISHFT	INTEGER(4)	INTEGER(4)
KISHFT [1]	INTEGER(8)	INTEGER(8)

[1] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BIT_SIZE, ISHFTC, ISHA, ISHC

Examples

ISHFT (2, 1) has the value 4.

ISHFT (2, -1) has the value 1.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, k(3), res2
i = 10 ! equal to 00001010
j = 10 ! equal to 00000000 00001010
res1 = ISHFT (i, 5)  ! returns 01000000 = 64
res2 = ISHFT (j, 5)  ! returns 00000001 01000000 =
                     ! 320

k = ISHFT((/3, 5, 1/), (/1, -1, 0/))  ! returns array
                                      ! /6, 2, 1/
```

ISHFTC

Elemental Intrinsic Function (Generic): Performs a circular shift of the rightmost bits.

Syntax

result = **ISHFTC** (*i*, *shift* [, *size*])

i
(Input) Must be of type integer.

shift
(Input) Must be of type integer. The absolute value of *shift* must be less than or equal to *size*.

size
(Optional; input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
IISHFTC	INTEGER(2)	INTEGER(2)
JISHFTC	INTEGER(4)	INTEGER(4)
KISHFTC [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BIT_SIZE, ISHFT, MVBITS

Examples

ISHFTC (4, 2, 4) has the value 1.

ISHFTC (3, 1, 3) has the value 6.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10  ! equal to 00001010
j = 10  ! equal to 00000000 00001010
res1 = ISHFTC (i, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 (left) and
                          ! returns 00001001 = 9
res1 = ISHFTC (i, -2, 3)  ! rotates the 3 rightmost
                          ! bits by -2 (right) and
                          ! returns 00001100 = 12
res2 = ISHFTC (j, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 and returns
                          ! 00000000 00001001 = 9
```

ISHL

Elemental Intrinsic Function (Generic): Logically shifts an integer left or right by the specified bits. Zeros are shifted in from the opposite end.

Syntax

result = **ISHL** (*i*, *shift*)

i
(Input) Must be of type integer. This argument is the value to be shifted.

shift
(Input) 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ISHA, ISHC, ISHFT, ISHFTC

Example

```
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
```

ISNAN

Elemental Intrinsic Function (Generic): Tests whether IEEE® real (S_floating and T_floating) numbers are Not-a-Number (NaN) values.

Syntax

result = **ISNAN** (*x*)

x
(Output) Must be of type real.

Results:

The result type is default logical. The result is .TRUE. if *x* is an IEEE NaN; otherwise, the result is .FALSE..

Examples

```
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..

ITIME

Portability Subroutine: Returns the time in numeric form.

Module: USE DFPORT

Syntax

CALL ITIME (*array*)

array
(Output) INTEGER(4). A rank one array with three elements used to store numeric time data.

The current time is returned in *array* in the order hour (*array*(1)), minute (*array*(2)), and second (*array*(3)).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, Portability Library

Example

```
USE DFPORT
 INTEGER(4) time_array(3)
 CALL ITIME (time_array)
 write(*,10) time_array
 10 format (1X,I2,':',I2,':',I2)
 END
```

JDATE

Portability Function: Returns an 8-character string with the Julian date in the form "yyddd". Three spaces terminate this string.

Module: USE DFPORT

Syntax

result = **JDATE** ( )

Results:

The result type is CHARACTER(8). The result is the Julian date, in the form YYDDD, followed by three spaces.

A Julian date is a five-digit number whose first two digits are the last two digits of the year, and whose final three represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on). For example, the Julian date for February 1, 1994 is 94032.

> **Warning:** The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE, DATE_AND_TIME, GETDAT, Portability Library

Example

```
! Sets julian to today's julian date
    USE DFPORT
    CHARACTER*8 julian
    julian = JDATE( )
```

KILL

Portability Function: Sends a signal to the process given by ID.

Module: USE DFPORT

Syntax

result = **KILL** (*pid*, *num*)

pid
(Input) INTEGER(4). ID of a process to be signaled.

num
(Input) INTEGER(4). Signal value. For the definition of signal values, see the **SIGNAL** function.

Results:

The result type is INTEGER(4). The result is zero if the call was successful; otherwise, an error code. Possible error codes are:

- **EINVAL:** The *signum* is not a valid signal number, or PID is not the same as getpid( ) and *signum* does not equal SIGKILL.
- **ESRCH:** The given PID could not be found.
- **EPERM:** The current process does not have permission to send a signal to the process given by PID.

Arbitrary signals can be sent only to the calling process (where *pid* = getpid( )). Other processes can send only the SIGKILL signal (*signum* = 9), and only if the calling process has permission.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RAISEQQ, SIGNALQQ, Portability Library

Example

```
USE DFPORT
integer(4) id_number, sig_val, istat
id_number=getpid( )
ISTAT = KILL (id_number, sig_val)
```

KIND

Inquiry Intrinsic Function (Generic): Returns the kind parameter of the argument.

Syntax

result = **KIND** (*x*)

x
(Input) 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SELECTED_INT_KIND, SELECTED_REAL_KIND, CMPLX, INT, REAL, LOGICAL, CHAR, Intrinsic Data Types

Examples

KIND (0.0) has the kind value of default real type.

KIND (12) has the kind value of default integer type.

The following shows another example:

```
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
```

LBOUND

Inquiry Intrinsic Function (Generic): Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.

Syntax

result = **LBOUND** (*array* [, *dim*])

array
(Input) Must be an array (of any data type). It must not be an allocatable array that is not allocated, or a disassociated pointer.

dim
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank *array*.

Results:

The result type is default integer. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: UBOUND

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.

The following shows another example:

```
REAL ARRAY (2:6, 8:14)
INTEGER LB(2), LBD
LB = LBOUND(ARRAY)           ! returns [2 8]
LBD = LBOUND(ARRAY, DIM = 2) ! returns 8
```

LCWRQQ (x86 only)

Run-Time Subroutine: Sets the value of the floating-point processor control word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL LCWRQQ (*controlword*)

controlword
(Input) INTEGER(2). Floating-point processor control word.

LCWRQQ performs the same function as the run-time subroutine SETCONTROLFPQQ and is provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFLIB
INTEGER(2) control
CALL SCWRQQ(control) ! get control word
!  Set control word to make processor round up
control = control .AND. (.NOT. FPCW$MCW_RC)  ! Clear
                                             ! control word with inverse
                                             ! of rounding control mask
control = control .OR. FPCW$UP  ! Set control word
                                ! to round up
CALL LCWRQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

LEADZ

Elemental Intrinsic Function (Generic): Returns the number of leading zero bits in an integer.

Syntax

result = **LEADZ** (*i*)

i
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 Model for Bit Data.

Example

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
```

LEN

Inquiry Intrinsic Function (Generic): Returns the length of a character expression.

Syntax

result = **LEN** (*string*)

string
(Input) Must be of type character; it can be scalar or array valued. (It can be an array of strings.)

Results:

The result is a scalar of type default integer. 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).

Specific Name	Argument Type	Result Type
LEN	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8) [1]
[1] INTEGER(8) is only available on Alpha processors.		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LEN_TRIM

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.

The following shows another example:

```
CHARACTER(11) STR(100)
INTEGER I
I = LEN (STR)  ! returns 11
I = LEN('A phrase with 5 trailing blanks. ')
               ! returns 37
```

LEN_TRIM

Elemental Intrinsic Function (Generic): Returns the length of the character argument without counting

trailing blank characters.

Syntax

result = **LEN_TRIM** (*string*)

string
(Input) Must be of type character.

Results:

The result type is default integer. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LEN, LNBLNK

Examples

In these examples, the symbol - represents a blank.

LEN_TRIM (’---C--D---’) has the value 7.

LEN_TRIM (’-----’) has the value 0.

The following shows another example:

```
INTEGER LEN1
LEN1 = LEN_TRIM (’ GOOD DAY ’) ! returns 9
LEN1 = LEN_TRIM (’ ’)          ! returns 0
```

LGE

Elemental Intrinsic Function (Generic): Determines if a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Syntax

result = **LGE** (*string_a*, *string_b*)

string_a
(Input) Must be of type character.

string_b
(Input) Must be of type character.

Results:

The result type is 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.		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LGT, LLE, LLT, ASCII and Key Code Charts

Examples

LGE (’ONE’, ’SIX’) has the value false.

LGE (’TWO’, ’THREE’) has the value true.

The following shows another example:

```
LOGICAL L
L = LGE(’ABC’,’ABD’)       ! returns .FALSE.
L = LGE (’AB’, ’AAAAAAAB’) ! returns .TRUE.
```

LGT

Elemental Intrinsic Function (Generic): Determines whether a string is lexically greater than another string, based on the ASCII collating sequence.

Syntax

result = **LGT** (*string_a*, *string_b*)

string_a
(Input) Must be of type character.

string_b
(Input) Must be of type character.

Results:

The result type is 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.		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LGE, LLE, LLT, ASCII and Key Code Charts

Examples

LGT (’TWO’, ’THREE’) has the value true.

LGT (’ONE’, ’FOUR’) has the value true.

The following shows another example:

```
LOGICAL L
L = LGT(’ABC’, ’ABC’)  ! returns .FALSE.
L = LGT(’ABC’, ’AABC’) ! returns .TRUE.
```

LINETO, LINETO_W

Graphics Function: Draws a line from the current graphics position up to and including the end point.

Module: USE DFLIB

Syntax

result = **LINETO** (*x*, *y*)
result = **LINETO_W** (*wx*, *wy*)

x, *y*
(Input) INTEGER(2). Viewport coordinates of end point.

wx, *wy*
(Input) REAL(8). Window coordinates of end point.

Results:

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0.

The line is drawn using the current graphics color, logical write mode, and line style. The graphics color is set with **SETCOLORRGB**, the write mode with **SETWRITEMODE**, and the line style with **SETLINESTYLE**.

If no error occurs, **LINETO** sets the current graphics position to the viewport point (*x*, *y*), and **LINETO_W** sets the current graphics position to the window point (*wx*, *wy*).

If you use **FLOODFILLRGB** to fill in a closed figure drawn with **LINETO**, the figure must be drawn with a solid line style. Line style is solid by default and can be changed with **SETLINESTYLE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

See Also: GETCURRENTPOSITION, GETLINESTYLE, GRSTATUS, MOVETO, POLYGON, SETLINESTYLE, SETWRITEMODE

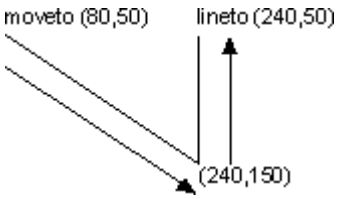
Example

This program draws the figure shown below.

```
! Build as QuickWin or Standard Graphics
USE DFLIB
INTEGER(2) status
TYPE (xycoord) xy

CALL MOVETO(INT2(80), INT2(50), xy)
status = LINETO(INT2(240), INT2(150))
status = LINETO(INT2(240), INT2(50))
END
```

Figure: Output of Program LINETO.FOR



LINETOAR

Graphics Function: Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array.

Module: USE DFLIB

Syntax

result = **LINETOAR** (loc(*fx*), loc(*fy*), loc(*tx*) loc(*ty*), *cnt*)

fx
(Input) INTEGER(2). From x viewport coordinate array.

fy
(Input) INTEGER(2). From y viewport coordinate array.

tx
(Input) INTEGER(2). To x viewport coordinate array.

ty
(Input) INTEGER(2). To y viewport coordinate array.

cnt
(Input) INTEGER(4). Length of each coordinate array; all should be the same size.

Results:

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the current graphics color, logical write mode, and line style. The graphics color is set with **SETCOLORRGB**, the write mode with **SETWRITEMODE**, and the line style with **SETLINESTYLE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

See Also: LINETO, LINETOAREX, LOC, SETCOLORRGB, SETLINESTYLE, SETWRITEMODE

Example

```
! Build for QuickWin or Standard Graphics
USE DFLIB
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) cnt, i
! load the points
do i = 1,3
  !from here
  fx(i) =20*i
  fy(i) =10
```

```
    !to there
    tx(i) =20*i
    ty(i) =60
  end do
  ! draw the lines all at once
  ! 3 white vertical lines in upper left corner
  result = LINETOAR(loc(fx),loc(fy),loc(tx),loc(ty), 3)
  end
```

LINETOAREX

Graphics Function: Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array. Each line is drawn with the specified graphics color and line style.

Module: USE DFLIB

Syntax

result = **LINETOAREX** (loc(*fx*), loc(*fy*), loc(*tx*) loc(*ty*), loc(*C*), loc(*S*),*cnt*)

fx
(Input) INTEGER(2). From x viewport coordinate array.

fy
(Input) INTEGER(2). From y viewport coordinate array.

tx
(Input) INTEGER(2). To x viewport coordinate array.

ty
(Input) INTEGER(2). To y viewport coordinate array.

C
(Input) INTEGER(4). Color array.

S
(Input) INTEGER(4). Style array.

cnt
(Input) INTEGER(4). Length of each coordinate array; also the length of the color array and style array. All of the arrays should be the same size.

Results:

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the specified graphics colors and line styles, and with the current write mode. The current write mode is set with **SETWRITEMODE**.

If the color has the #80000000 bit set, the color is an RGB color; otherwise, the color is a palette

color.

The styles are as follows from wingdi.h:

```
SOLID            0
DASH             1       /* -------  */
DOT              2       /* .......  */
DASHDOT          3       /* _._._._  */
DASHDOTDOT       4       /* _.._.._  */
NULL             5
```

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

See Also: LINETO, LINETOAR, LOC, SETWRITEMODE

Example

```
! Build for QuickWin or Standard Graphics
USE DFLIB
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) C(3),S(3),cnt,i,color

color = #000000FF

! load the points
do i = 1,3
   S(i) = 0  ! all lines solid
   C(i) = IOR(#80000000,color)
   color = color*256 ! pick another of RGB
   if(IAND(color,#00FFFFFF).eq.0) color = #000000FF
   !from here
   fx(i) =20*i
   fy(i) =10
   !to there
   tx(i) =20*i
   ty(i) =60
 end do
 ! draw the lines all at once
 ! 3 vertical lines in upper left corner, Red, Green, and Blue
 result = LINETOAREX(loc(fx),loc(fy),loc(tx),loc(ty),loc(C),loc(S),3)
 end
```

LLE

Elemental Intrinsic Function (Generic): Determines whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Syntax

result = **LLE** (*string_a*, *string_b*)

string_a

(Input) Must be of type character.

string_b
(Input) Must be of type character.

Results:

The result type is 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.		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LGE, LGT, LLT, ASCII and Key Code Charts

Examples

LLE ('TWO', 'THREE') has the value false.

LLE ('ONE', 'FOUR') has the value false.

The following shows another example:

```
LOGICAL L
L = LLE(’ABC’, ’ABC’)   ! returns .TRUE.
L = LLE(’ABC’, ’AABCD’) ! returns .FALSE.
```

LLT

Elemental Intrinsic Function (Generic): Determines whether a string is lexically less than another string, based on the ASCII collating sequence.

Syntax

result = **LLT** (*string_a*, *string_b*)

string_a
(Input) Must be of type character.

string_b
(Input) Must be of type character.

Results:

The result type is 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.		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LGE, LGT, LLE, ASCII and Key Code Charts

Examples

LLT (’ONE’, ’SIX’) has the value true.

LLT (’ONE’, ’FOUR’) has the value false.

The following shows another example:

```
LOGICAL L
L = LLT (’ABC’, ’ABC’)     ! returns .FALSE.
L = LLT (’AAXYZ’, ’ABCDE’) ! returns .TRUE.
```

LNBLNK

Portability Function: Locates the position of the last nonblank character in a string.

Module: USE DFPORT

Syntax

result = **LNBLNK** (*string*)

string
(Input) Character*(*). String to be searched. Cannot be an array.

Results:

The result type is INTEGER(4). The result is the index of the last nonblank character in *string*.

LNBLNK is very similar to the intrinsic function **LEN_TRIM,** except that *string* cannot be an array.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LEN_TRIM, Portability Library

Example

```
USE DFPORT
integer(4) p
p = LNBLNK(' GOOD DAY ') ! returns 9
p = LNBLNK(' ')          ! returns 0
```

LOADIMAGE, LOADIMAGE_W

Graphics Function: Reads an image from a Windows bitmap file and displays it at a specified location.

Module: USE DFLIB

Syntax

result = **LOADIMAGE** (*filename*, *xcoord*, *ycoord*)
result = **LOADIMAGE_W** (*filename*, *wxcoord*, *wycoord*)

filename
(Input) Character*(*). Path of the bitmap file.

xcoord, *ycoord*
(Input) INTEGER(4). Viewport coordinates for upper-left corner of image display.

wxcoord, *wycoord*
(Input) REAL(8). Window coordinates for upper-left corner of image display.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The image is displayed with the colors in the bitmap file. If the color palette in the bitmap file is different from the current system palette, the current palette is discarded and the bitmap's palette is loaded.

LOADIMAGE specifes the screen placement of the image in viewport coordinates.
LOADIMAGE_W specifies the screen placement of the image in window coordinates.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SAVEIMAGE, SAVEIMAGE_W

LOC

Inquiry Intrinsic Function (Generic): Returns the internal address of a storage item. This function cannot be passed as an actual argument.

Syntax

result = **LOC** (*x*)

x
(Input) 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 type is INTEGER(4) on Intel processors; INTEGER(8) on Alpha processors. 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.

LOC performs the same function as the %LOC built-in function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
! Mixed language example passing Integer Pointer to C

! Fortran main program
      INTERFACE
         SUBROUTINE Ptr_Sub (p)
!DEC$      ATTRIBUTES C, ALIAS:'_Ptr_Sub' :: Ptr_Sub
           INTEGER p
         END SUBROUTINE Ptr_Sub
      END INTERFACE

      REAL A[10], VAR[10]
      POINTER (p, VAR)  ! VAR is the pointer-based
                        ! variable, p is the integer
                        ! pointer
```

```
      p = LOC(A)

      CALL Ptr_Sub (p)
      WRITE(*,*) "A(4) = ", A(4)
      END

! C subprogram
void Ptr_Sub (int *p)
{
  float a[10];
  a[3] = 23.5;
  *p = a;
}
```

%LOC

Built-in Function: Computes the internal address of a storage item.

Syntax

result = **%LOC** (*a*)

a
(Input) 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 (INTEGER(4) on Windows NT and Windows 95 systems; INTEGER(8) on OpenVMS and DIGITAL UNIX systems) value that represents the location of the given argument. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

LOG

Elemental Intrinsic Function (Generic): Returns the natural logarithm of the argument.

Syntax

result = **LOG** (*x*)

x
(Input) 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 omega in the range -pi < omega <= pi. The imaginary part of the result is pi 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 [2]	REAL(16)	REAL(16)
CLOG [1]	COMPLEX(4)	COMPLEX(4)
CDLOG [3]	COMPLEX(8)	COMPLEX(8)

[1] The setting of compiler option /real_size can affect ALOG and CLOG.
[2] VMS and U*X
[3] This function can also be specified as ZLOG.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EXP, LOG10

Examples

LOG (8.0) has the value 2.079442.

LOG (25.0) has the value 3.218876.

The following shows another example:

```
REAL r
r = LOG(10.0)  ! returns 2.302585
```

LOG10

Elemental Intrinsic Function (Generic): Returns the common logarithm of the argument.

Syntax

result = **LOG10** (*x*)

x
(Input) 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 [2]	REAL(16)	REAL(16)

[1] The setting of compiler option /real_size can affect ALOG10.
[2] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LOG

Examples

LOG10 (8.0) has the value 0.9030900.

LOG10 (15.0) has the value 1.176091.

The following shows another example:

```
REAL r
r = LOG10(10.0)  ! returns 1.0
```

LOGICAL Function

Elemental Intrinsic Function (Generic): Converts the logical value of the argument to a logical value with different kind parameters.

Syntax

result = **LOGICAL**(*l* [, *kind*])

l
(Input) Must be of type logical.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is 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 option /integer_size can affect this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CMPLX, INT, REAL

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 default integer.

LOGICAL

Statement: Specifies the LOGICAL data type.

Syntax

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

n
Is kind 1, 2, 4, or 8. Kind 8 is only available on Alpha processors.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Logical Data Types, Logical Constants, Data Types, Constants, and Variables

Example

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (2), SAVE :: doit, dont=.FALSE.
LOGICAL switch

! An equivalent declaration is:
```

```
LOGICAL flag1, flag2
LOGICAL (2) doit, dont=.FALSE.
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

LONG

Portability Function: Returns an INTEGER(2) value as an INTEGER(4) type.

Module: USE DFPORT

Syntax

result = **LONG** (*int2*)

int2
(Input) INTEGER(2). Value to be converted.

Results:

The result type is INTEGER(4). The result is the value of *int2* with type INTEGER(4). The upper 16 bits of the result are zeros and the lower 16 are equal to *int2*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INT, KIND, Portability Library

LSHIFT

Elemental Intrinsic Function: Shifts the bits in an integer left by a specified number of positions. For more information, see ISHFT.

LSTAT

Portability Function: Returns detailed information about a file.

Module: USE DFPORT

Syntax

result = **LSTAT** (*name*, *statb*)

name
(Input) Character*(*). Name of the file to examine.

statb

(Output) INTEGER(4). One-dimensional array with a size of 12. See **STAT** for the possible values returned in *statb*.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code (see IERRNO.

LSTAT returns detailed information about the file named in *name*. In this implementation, **LSTAT** returns exactly the same information as **STAT** (because there are no symbolic links). **STAT** is the preferred function.

INQUIRE also provides information about file properties.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INQUIRE, GETFILEINFOQQ, STAT, FSTAT, Portability Library

Example

```
USE DFPORT
INTEGER(4) info_array(12), istatus
character*20 file_name
print *, "Enter name of file to examime: "
read *, file_name
ISTATUS = LSTAT (file_name, info_array)
if (.NOT. ISTATUS) then
   print *, info_array
else
   print *, 'Error ',istatus
end if
```

LTIME

Portability Subroutine: Returns the components of the local time zone time in a nine-element array.

Module: USE DFPORT

Syntax

CALL **LTIME** (*time*, *array*)

time
(Input) Integer(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

array
(Output) Integer(4). One-dimensional array with 9 elements to contain local date and time data derived from *time*.

The elements of *array* are returned as follows:

Element of *array*	Data returned
array(1)	Seconds (0 - 59)
array(2)	Minutes (0 - 59)
array(3)	Hours (0 - 23)
array(4)	Day of month (1 - 31)
array(5)	Month (0 - 11)
array(6)	Year number in century (0 - 99)
array(7)	Day of week (0 - 6, where 0 is Sunday)
array(8)	Day of year (1 - 365)
array(9)	1 if daylight saving time is in effect; otherwise, 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, Portability Library

Example

```
USE DFPORT
INTEGER(4) input_time, time_array
!   find number of seconds since 1/1/70
input_time=TIME()
!   convert number of seconds to time array
CALL LTIME (input_time, time_array)
PRINT *, time_array
```

MAKEDIRQQ

Run-Time Function: Creates a new directory with a specified name.

Module: USE DFLIB

Syntax

result = **MAKEDIRQQ** (*dirname*)

dirname
(Input) Character*(*). Name of directory to be created.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

MAKEDIRQQ can create only one directory at a time. You cannot create a new directory and a subdirectory below it in a single command. **MAKEDIRQQ** does not translate path delimiters. You can use either slash (/) or backslash (\) as valid delimiters.

If an error occurs, call **GETLASTERRORQQ** to retrieve the error message. Possible errors include:

- **ERR$ACCESS**: The directory was not created. The given name is the name of an existing file, directory, or device.
- **ERR$NOENT**: The path was not found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DELDIRQQ, CHANGEDIRQQ, GETLASTERRORQQ

Example

```
USE DFLIB
LOGICAL(4) result
result = MAKEDIRQQ('mynewdir')
IF (result) THEN
   WRITE (*,*) 'New subdirectory successfully created'
ELSE
   WRITE (*,*) 'Failed to create subdirectory'
END IF
END
```

MALLOC

Elemental Intrinsic Function (Specific): Allocates a block of memory. This specific function cannot be passed as an actual argument.

Syntax

result = **MALLOC** (*i*)

i
(Input) Must be of type INTEGER(4). This value is the size (in bytes) of memory to be allocated.

Results:

The result type is INTEGER(4) on Intel processors; INTEGER(8) on Alpha processors. The result is the starting address of the allocated memory. The memory allocated can be freed by using the FREE intrinsic function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
  INTEGER(4) addr, size
  size = 1024         ! size in bytes
  addr = MALLOC(size) ! allocate the memory
  CALL FREE(addr)     ! free it
  END
```

MAP...END MAP

Statement: Specifies mapped field declarations that are part of a **UNION** declaration within a **STRUCTURE** statement. For more information, see UNION...END UNION.

Example

```
UNION
  MAP
     CHARACTER*20 string
  END MAP
  MAP
     INTEGER*2 number(10)
  END MAP
END UNION

UNION
  MAP
     RECORD /Cartesian/ xcoord, ycoord
  END MAP
  MAP
     RECORD /Polar/ length, angle
  END MAP
END UNION
```

MATHERRQQ (x86 only)

Run-Time Subroutine: Handles run-time math errors. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL MATHERRQQ (*name*, *nlen*, *info*, *retcode*)

name
(Output) Character*(*). Name of the function causing the error. The parameter *name* is a typeless version of the function called. For example, if an error occurs in a **SIN** function, the name will be returned as sin for real arguments and csin for complex arguments even though the function may have actually been called with an alternate name such as **DSIN** or **CDSIN**, or with **SIN** and complex arguments.

nlen
(Output) INTEGER(2). Length of *name*.

info
(Output) Structure. Record containing data about the error. The MTH$E_INFO structure is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
STRUCTURE /MTH$E_INFO/
  INTEGER*4 ERRCODE      ! One of the MTH$ values below
  INTEGER*4 FTYPE        ! One of the TY$ values below
  UNION
  MAP
    REAL*4 R4ARG1        ! INPUT: First argument
    CHARACTER*12 R4FILL1
    REAL*4 R4ARG2        ! INPUT: Second argument (if any)
    CHARACTER*12 R4FILL2
    REAL*4 R4RES         ! OUTPUT : Desired result
    CHARACTER*12 R4FILL3
  END MAP
  MAP
    REAL*8 R8ARG1        ! INPUT : First argument
    CHARACTER*8 R8FILL1
    REAL*8 R8ARG2        ! INPUT : Second argument (if any)
    CHARACTER*8 R8FILL2
    REAL*8 R8RES         ! OUTPUT : Desired result
    CHARACTER*8 R8FILL3
  END MAP
  MAP
    COMPLEX*8 C8ARG1     ! INPUT : First argument
    CHARACTER*8 C8FILL1
    COMPLEX*8 C8ARG2     ! INPUT : Second argument (if any)
    CHARACTER*8 C8FILL2
    COMPLEX*8 C8RES      ! OUTPUT : Desired result
    CHARACTER*8 C8FILL1
  END MAP
  MAP
    COMPLEX*16 C16ARG1    ! INPUT : First argument
    COMPLEX*16 C16ARG2    ! INPUT : Second argument (if any)
    COMPLEX*16 C16RES     ! OUTPUT : Desired result
```

```
    END MAP
    END UNION
  END STRUCTURE
```

retcode
(Output) INTEGER(2). Return code passed back to the run-time library. The value of *retcode* should be set by the user's **MATHERRQQ** routine to indicate whether the error was resolved. Set this value to 0 to indicate that the error was not resolved and that the program should fail with a run-time error. Set it to any nonzero value to indicate that the error was resolved and the program should continue.

If you are not compiling with full optimization (using the /Ox compiler option), errors in math functions generate a call to the **MATHERRQQ** subroutine. You can write a **MATHERRQQ** function that resolves the error or takes other appropriate action based on arguments passed to the function. If you do not provide your own **MATHERRQQ** function, a default **MATHERRQQ** provided with the library terminates the process.

Under the ANSI definition of Fortran, there is no handling of math errors. The programmer is responsible for making sure that arguments to math intrinsics are valid. If they are not valid, the result is undefined. Handling of math errors in math debug mode is a language extension. This mode provides more safety, but the performance of math functions is significantly slower.

The ERRCODE element in the MTH$E_INFO structure specifies the type of math error that occurred, and can have one of the following values:

Value	Meaning
MTH$E_DOMAIN	Argument domain error
MTH$E_OVERFLOW	Overflow range error
MTH$E_PLOSS	Partial loss of significance
MTH$E_SINGULARITY	Argument singularity
MTH$E_TLOSS	Total loss of significance
MTH$E_UNDERFLOW	Underflow range error

The FTYPE element of the *info* structure identifies the data type of the math function as TY$REAL4, TY$REAL8, TY$CMPLX4, or TY$CMPLX8. Internally, REAL(4) and COMPLEX(4) arguments are converted to REAL(8) and COMPLEX(8). This means that the corresponding mapped sections of the structure are never used.

In general, a **MATHERRQQ** function should test the FTYPE value and take separate action for TY$REAL8 or TY$CMPLX8 using the appropriate mapped values. If you want to resolve the error, set the R8RES or C8RES field to an appropriate value such as 0.0. You can do calculations within the **MATHERRQQ** function using the approprate ARG1 and ARG2 fields, but avoid doing any calculations that would cause an error resulting in another call to **MATHERRQQ**.

Note: You cannot use **MATHERRQQ** in DLLs or in a program that links with a DLL.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS LIB

Example

See the example `MATHERRQQ` in Handling Run-Time Math Exceptions in the *Programmer's Guide*.

MATMUL

Transformational Intrinsic Function (Generic): Performs matrix multiplication of numeric or logical matrices.

Syntax

result = **MATMUL** (*matrix_a*, *matrix_b*)

matrix_a
(Input) Must be an array of rank one or two. It must be of numeric (integer, real, or complex) or logical type.

matrix_b
(Input) 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 Conversion Rules for Numeric Assignment Statements. 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*)).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: TRANSPOSE, PRODUCT

Examples

A is matrix

```
[ 2  3  4 ]
[ 3  4  5 ],
```

B is matrix

```
[ 2  3 ]
[ 3  4 ]
[ 4  5 ],
```

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

```
[ 29  38 ]
[ 38  50 ].
```

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).

The following shows another example:

```
INTEGER a(2,3), b(3,2), c(2), d(3), e(2,2), f(3), g(2)
a = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  a is   1 3 5
!         2 4 6
b = RESHAPE((/1, 2, 3, 4, 5, 6/), (/3, 2/))
!  b is   1 4
!         2 5
!         3 6
c = (/1, 2/)      ! c is [1 2]
d = (/1, 2, 3/)   ! d is [1 2 3]

e = MATMUL(a, b)    ! returns 22 49
                    !         28 64

f = MATMUL(c,a)   ! returns [5 11 17]
g = MATMUL(a,d)   ! returns [22 28]
WRITE(*,*) e, f, g
END
```

MAX

Elemental Intrinsic Function (Generic): Returns the maximum value of the arguments.

Syntax

result = **MAX** (*a1*, *a2* [, *a3*] ...)

a1, *a2*, *a3*
(Input) 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 type is integer. For **AMAX0**, **AIMAX0**, **AJMAX0**, and **AKMAX0**, the result type is 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 [3, 4]	INTEGER(4)	REAL(4)
KMAX0 [5]	INTEGER(8)	INTEGER(8)
AKMAX0 [5]	INTEGER(8)	REAL(4)
IMAX1	REAL(4)	INTEGER(2)
MAX1 [4, 6, 7]	REAL(4)	INTEGER(4)
KMAX1 [5]	REAL(4)	INTEGER(8)
AMAX1 [8]	REAL(4)	REAL(4)
DMAX1	REAL(8)	REAL(8)
QMAX1 [9]	REAL(16)	REAL(16)

[1] These specific functions cannot be passed as actual arguments.
[2] Or JMAX0.
[3] Or AJMAX0. AMAX0 is the same as REAL (MAX).
[4] In Fortran 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.
[5] Alpha only
[6] Or JMAX1. MAX1 is the same as INT (MAX).
[7] The setting of compiler option /integer_size can affect MAX1.
[8] The setting of compiler option /real_size can affect AMAX1.
[9] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MIN

Examples

MAX (2.0, -8.0, 6.0) has the value 6.0.

MAX (14, 32, -50) has the value 32.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MAX(5, 6, 7)           ! returns 7
m2 = MAX1(5.7, 3.2, -8.3)   ! returns 5
r1 = AMAX0(5, 6, 7)         ! returns 7.0
r2 = AMAX1(6.4, -12.2, 4.9) ! returns 6.4
```

MAXEXPONENT

Inquiry Intrinsic Function (Generic): Returns the maximum exponent in the model representing the same type and kind parameters as the argument.

Syntax

result = **MAXEXPONENT** (*x*)

x
(Input) 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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MINEXPONENT

Examples

```
REAL(4) x
INTEGER i
i = MAXEXPONENT(x)    ! returns 128.
```

MAXLOC

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **MAXLOC** (*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer or real.

dim
(Optional; input) 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
(Optional; input) Must be a logical array that is conformable with *array*.

Results:

The result is an array of type default integer.

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*.
- 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, the value of the result is undefined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAXVAL, MINLOC, MINVAL

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

```
[  4    0   -3    2 ]
[  3    1   -2    6 ]
[ -1   -4    5   -5 ].
```

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.

The following shows another example:

```
INTEGER i, max
INTEGER i, maxl(1)
INTEGER array(3, 3)
INTEGER, ALLOCATABLE :: AR1(:)
! put values in array
array = RESHAPE((/7, 9, -1, -2, 5, 0, 3, 6, 9/),     &
                (/3, 3/))
! array is  7 -2 3
!           9 5 6
!          -1 0 9
```

```
i = SIZE(SHAPE(array))   ! Get number of dimensions
                         ! in array
ALLOCATE ( AR1(i))       ! Allocate AR1 to number
                         ! of dimensions in array
AR1 = MAXLOC (array, MASK = array .LT. 7) ! Get
                         ! the location (subscripts) of
                         ! largest element less than 7
                         ! in array

!
! MASK = array .LT. 7 creates a mask array the same
! size and shape as array whose elements are .TRUE. if
! the corresponding element in array is less than 7,
! and .FALSE. if it is not. This mask causes MAXLOC to
! return the index of the element in array with the
! greatest value less than 7.
!
! array is  7 -2 3 and MASK=array .LT. 7 is  F T T
!           9  5 6                           F T T
!          -1  0 9                           T T F
! and AR1 = MAXLOC(array, MASK = array .LT. 7) returns
! (2, 3), the location of the element with value 6

maxl = MAXLOC((/1, 4, 3, 4/))    ! returns 2, the first
                                 ! occurrence of maximum
END
```

MAXVAL

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **MAXVAL** (*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer or real.

dim
(Optional; input) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *array*.

mask
(Optional; input) 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 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAXLOC, MINVAL, MINLOC

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

```
[ 2  3  4 ]
[ 5  6  7 ].
```

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.

The following shows another example:

```
INTEGER array(2,3), i(2), max
```

```
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/),(/2, 3/))
! array is   1 5 3
!            4 2 6
i = SHAPE(array)      ! i = [2 3]
ALLOCATE (AR1(i(2)))  ! dimension AR1 to the number of
                      ! elements in dimension 2
                      ! (a column) of array
ALLOCATE (AR2(i(1)))  ! dimension AR2 to the number of
                      ! elements in dimension 1
                      ! (a row) of array
max = MAXVAL(array, MASK = array .LT. 4) ! returns 3
AR1 = MAXVAL(array, DIM = 1)  ! returns [ 4 5 6 ]
AR2 = MAXVAL(array, DIM = 2)  ! returns [ 5 6 ]
END
```

MBCharLen

NLS Function: Returns the length, in bytes, of the first character in a multibyte-character string.

Module: USE DFNLS

Syntax

result = **MBCharLen** (*string*)

string
(Input) Character*(*). String containing the character whose length is to be determined. Can contain multibyte characters.

Results:

The result type is INTEGER(4). The result is the number of bytes in the first character contained in *string*. Returns 0 if *string* has no characters (is length 0).

MBCharLen does not test for multibyte character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBCurMax, MBLead, MBLen, MBLen_Trim

MBConvertMBToUnicode

NLS Function: Converts a multibyte-character string from the current codepage to a Unicode string.

Module: USE DFNLS

Syntax

result = **MBConvertMBToUnicode** (*mbstr*, *unicodestr* [, *flags*])

mbstr
(Input) Character*(*). Multibyte codepage string to be converted.

unicodestr
(Output) INTEGER(2). Array of integers that is the translation of the input string into Unicode.

flags
(Optional; input) INTEGER(4). If specified, modifies the string conversion. If *flags* is omitted, the value NLS$Precomposed is used. Available values (defined in DFNLS.F90) are:

- **NLS$Precomposed**: Use precomposed characters always. (default)
- **NLS$Composite**: Use composite wide characters always.
- **NLS$UseGlyphChars**: Use glyph characters instead of control characters.
- **NLS$ErrorOnInvalidChars**: Returns - 1 if an invalid input character is encountered.

The flags NLS$Precomposed and NLS$Composite are mutually exclusive. You can combine NLS$UseGlyphChars with either NLS$Precomposed or NLS$Composite using an inclusive OR (IOR or OR).

Results:

The result type is INTEGER(4). If no error occurs, returns the number of bytes written to *unicodestr* (bytes are counted, not characters), or the number of bytes required to hold the output string if *unicodestr* has zero size. If the *unicodestr* array is bigger than needed to hold the translation, the extra elements are set to space characters. If *unicodestr* has zero size, the function returns the number of bytes required to hold the translation and nothing is written to *unicodestr*.

If an error occurs, one of the following negative values is returned:

- **NLS$ErrorInsufficentBuffer**: The *unicodestr* argument is too small, but not zero size so that the needed number of bytes would be returned.
- **NLS$ErrorInvalidFlags**: The *flags* argument has an illegal value.
- **NLS$ErrorInvalidCharacter**: A character with no Unicode translation was encountered in *mbstr*. This error can occur only if the **NLS$InvalidCharsError** flag was used in *flags*.

By default, or if *flags* is set to NLS$Precomposed, the function MBConvertMBToUnicode attempts to translate the multibyte codepage string to a precomposed Unicode string. If a precomposed form does not exist, the function attempts to translate the codepage string to a composite form.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBConvertUnicodeToMB

MBConvertUnicodeToMB

NLS Function: Converts a Unicode string to a multibyte-character string from the current codepage.

Module: USE DFNLS

Syntax

result = **MBConvertUnicodeToMB** (*unicodestr*, *mbstr* [, *flags*])

unicodestr
(Input) INTEGER(2). Array of integers holding the Unicode string to be translated.

mbstr
(Output) Character*(*). Translation of Unicode string into multibyte character string from the current codepage.

flags
(Optional; input) INTEGER(4). If specified, argument to modify the string conversion. If *flags* is omitted, no extra checking of the conversion takes place. Available values (defined in DFNLS.F90) are:

- **NLS$CompositeCheck**: Convert composite characters to precomposed.
- **NLS$SepChars**: Generate separate characters.
- **NLS$DiscardDns**: Discard nonspacing characters.
- **NLS$DefaultChars**: Replace exceptions with default character.

The last three flags (NLS$SepChars, NLS$DiscardDns, and NLS$DefaultChars) are mutually exclusive and can be used only if NLS$CompositeCheck is set, in which case one (and only one) of them is combined with NLS$CompositeCheck using an inclusive OR (IOR or OR). These flags determine what translation to make when there is no precomposed mapping for a base character/nonspace character combination in the Unicode wide character string. The default (IOR(NLS$CompositeCheck, NLS$SepChars)) is to generate separate characters.

Results:

The result type is INTEGER(4). If no error occurs, returns the number of bytes written to *mbstr* (bytes are counted, not characters), or the number of bytes required to hold the output string if *mbstr* has zero length. If *mbstr* is longer than the translation, it is blank-padded. If *mbstr* is zero length, the function returns the number of bytes required to hold the translation and nothing is written to *mbstr*.

If an error occurs, one of the following negative values is returned:

- **NLS$ErrorInsufficentBuffer**: The *mbstr* argument is too small, but not zero length so that the needed number of bytes is returned.
- **NLS$ErrorInvalidFlags**: The *flags* argument has an illegal value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBConvertMBToUnicode

MBCurMax

NLS Function: Returns the longest possible multibyte character length, in bytes, for the current codepage.

Module: USE DFNLS

Syntax

result = **MBCurMax** ( )

Results:

The result type is INTEGER(4). The result is the longest possible multibyte character, in bytes, for the current codepage.

The MBLenMax parameter, defined in the module DFNLS.F90, is the longest length, in bytes, of any character in any codepage installed on the system.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBCharLen

MBINCHARQQ

NLS Function: Performs the same function as **INCHARQQ** except that it can read a single multibyte character at once, and it returns the number of bytes read as well as the character.

Module: USE DFNLS

Syntax

result = **MBINCHARQQ** (*string*)

string
(Output) CHARACTER(MBLenMax). String containing the read characters, padded with blanks up to the length MBLenMax. The MBLenMax parameter, defined in the module DFNLS.F90 (in \DF98\INCLUDE), is the longest length, in bytes, of any character in any codepage installed on the system.

Results:

The result type is INTEGER(4). The result is the number of characters read.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INCHARQQ, MBCurMax, MBCharLen, MBLead

MBINDEX

NLS Function: Performs the same function as **INDEX** except that the strings manipulated can contain multibyte characters.

Module: USE DFNLS

Syntax

result = **MBINDEX** (*string*, *substring* [, *back*])

string
(Input) CHARACTER*(*). String to be searched for the presence of *substring*. Can contain multibyte characters.

substring
(Input) CHARACTER*(*). Substring whose position within *string* is to be determined. Can contain multibyte characters.

back
(Optional; input) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results:

The result type is INTEGER(4). If *back* is omitted or is .FALSE., returns the leftmost position in *string* that contains the start of *substring*. If *back* is .TRUE., returns the rightmost position in *string* which contains the start of *substring*. If *string* does not contain *substring*, returns 0. If *substring* occurs more than once, returns the starting position of the first occurrence ("first" is determined by the presence and value of *back*).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INDEX, MBSCAN, MBVERIFY

MBJISToJMS and MBJMSToJIS

NLS Functions: Convert Japan Industry Standard (JIS) characters to Microsoft Kanji (JMS) characters or converts JMS characters to JIS characters.

Module: USE DFNLS

Syntax

result = **MBJISToJMS** (*char*)
result = **MBJMSToJIS** (*char*)

char
(Input) CHARACTER(2). JIS or JMS character to be converted.

A JIS character is converted only if the lead and trail bytes are in the hexadecimal range 21 through 7E.

A JMS character is converted only if the lead byte is in the hexadecimal range 81 through 9F or E0 through FC, and the trail byte is in the hexadecimal range 40 through 7E or 80 through FC.

Results:

The result type is CHARACTER(2). **MBJISToJMS** returns a Microsoft Kanji (Shift JIS or JMS) character. **MBJMSToJIS** returns a Japan Industry Standard (JIS) character.

Only computers with Japanese installed as one of the available languages can use the **MBJISToJMS** and **MBJMSToJIS** conversion functions.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSEnumLocales, NLSEnumCodepages, NLSGetLocale, NLSSetLocale

MBLead

NLS Function: Determines whether a given character is the lead (first) byte of a multibyte character sequence.

Module: USE DFNLS

Syntax

result = **MBLead** (*char*)

char

(Input) CHARACTER(1). Character to be tested for lead status.

Results:

The result type is LOGICAL(4). The result is .TRUE. if *char* is the first character of a multibyte character sequence; otherwise, .FALSE..

MBLead only works stepping forward through a whole multibyte character string. For example:

```
       DO i = 1, LEN(str)  ! LEN returns the number of bytes, not the
                           ! number of characters in str
          WRITE(*, 100) MBLead (str(i:i))
       END DO
 100   FORMAT (L2, \)
```

MBLead is passed only one character at a time and must start on a lead byte and step through a string to establish context for the character. **MBLead** does not correctly identify a nonlead byte if it is passed only the second byte of a multibyte character because the status of lead byte or trail byte depends on context.

The function **MBStrLead** is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context. So, **MBStrLead** can be much slower than **MBLead** (up to *n* times slower, where *n* is the length of the string).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBStrLead, MBCharLen

MBLen

NLS Function: Returns the number of characters in a multibyte-character string, including trailing blanks.

Module: USE DFNLS

Syntax

result = **MBLen** (*string*)

string
(Input) CHARACTER*(*). String whose characters are to be counted. Can contain multibyte characters.

Results:

The result type is INTEGER(4). The result is the number of characters in *string*.

MBLen recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBLen_Trim, MBStrLead

MBLen_Trim

NLS Function: Returns the number of characters in a multibyte-character string, not including trailing blanks.

Module: USE DFNLS

Syntax

result = **MBLen_Trim** (*string*)

string
(Input) Character*(*). String whose characters are to be counted. Can contain multibyte characters.

Results:

The result type is INTEGER(4). The result is the number of characters in *string* minus any trailing blanks (blanks are bytes containing character 32 (hex 20) in the ASCII collating sequence).

MBLen_Trim recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBLen, MBStrLead

MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE

NLS Functions: Perform the same functions as **LGE**, **LGT**, **LLE**, **LLT** and the logical operators .EQ. and .NE. except that the strings being compared can include multibyte characters, and optional flags can modify the comparison.

Module: USE DFNLS

Syntax

result = **MBLGE** (*string_a*, *string_b*, [*flags*])
result = **MBLGT** (*string_a*, *string_b*, [*flags*])
result = **MBLLE** (*string_a*, *string_b*, [*flags*])
result = **MBLLT** (*string_a*, *string_b*, [*flags*])
result = **MBLEQ** (*string_a*, *string_b*, [*flags*])
result = **MBLNE** (*string_a*, *string_b*, [*flags*])

string_a, *string_b*
(Input) Character*(*). Strings to be compared. Can contain multibyte characters.

flags
(Optional; input) INTEGER(4). If specified, determines which character traits to use or ignore when comparing strings. You can combine several flags using an inclusive OR (IOR or OR). There are no illegal combinations of flags, and the functions may be used without flags, in which case all flag options are turned off. The available values (defined in DFNLS.F90) are:

- **NLS$MB_IgnoreCase**: Ignore case.
- **NLS$MB_IgnoreNonspace**: Ignore nonspacing characters (this flag removes Japanese accent characters if they exist).
- **NLS$MB_IgnoreSymbols**: Ignore symbols.
- **NLS$MB_IgnoreKanaType**: Do not differentiate between Japanese Hiragana and Katakana characters (corresponding Hiragana and Katakana characters will compare as equal).
- **NLS$MB_IgnoreWidth**: Do not differentiate between a single-byte character and the same character as a double byte.
- **NLS$MB_StringSort**: Sort all symbols at the beginning, including the apostrophe and hyphen (See last paragraph below).

Results:

The result type is LOGICAL(4). Comparisons are made using the current locale, not the current codepage. The codepage used is the default for the language/country combination of the current locale.

MBLGE returns .TRUE. if the strings are equal or *string_a* comes last in the collating sequence. Otherwise, it returns .FALSE..

MBLGT returns .TRUE. if *string_a* comes last in the collating sequence. Otherwise, it returns .FALSE..

MBLLE returns .TRUE. if the strings are equal or *string_a* comes first in the collating sequence. Otherwise, it returns .FALSE..

MBLLT returns .TRUE. if *string_a* comes first in the collating sequence. Otherwise, it returns .FALSE..

MBLEQ returns .TRUE. if the strings are equal in the collating sequence. Otherwise, it returns .FALSE..

MBLNE returns .TRUE. if the strings are not equal in the collating sequence. Otherwise, it returns .FALSE..

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, then the return value indicates that the longer string is greater.

If *flags* is invalid, the functions return .FALSE..

If the strings supplied contain Arabic Kashidas, the Kashidas are ignored during the comparison. Therefore, if the two strings are identical except for Kashidas within the strings, the functions return a value indicating they are "equal" in the collation sense, though not necessarily identical.

When not using the NLS$MB_StringSort flag, the hyphen and apostrophe are special symbols and are treated differently than others. This is to ensure that words like coop and co-op stay together within a list. All symbols, except the hyphen and apostrophe, sort before any other alphanumeric character. If you specify the NLS$MB_StringSort flag, hyphen and apostrophe sort at the beginning also.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LGE, LGT, LLE, LLT

MBNext

NLS Function: Returns the position of the first lead byte or single-byte character immediately following the given position in a multibyte-character string.

Module: USE DFNLS

Syntax

result = **MBNext** (*string*, *position*)

string
(Input) Character*(*). String to be seached for the first lead byte or single-byte character after the current position. Can contain multibyte characters.

position
(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or a single-byte character. Cannot be the position of a trail (second) byte of a multibyte character.

Results:

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately following the position given in *position*, or 0 if no following first byte

is found in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBPrev

MBPrev

NLS Function: Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multibyte-character string.

Module: USE DFNLS

Syntax

result = **MBPrev** (*string*, *position*)

string
(Input) Character*(*). String to be seached for the first lead byte or single-byte character before the current position. Can contain multibyte characters.

position
(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or single-byte character. Cannot be the position of the trail (second) byte of a multibyte character.

Results:

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately preceding the position given in *position*, or 0 if no preceding first byte is found in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBNext

MBSCAN

NLS Function: Performs the same function as **SCAN** except that the strings manipulated can contain multibyte characters.

Module: USE DFNLS

Syntax

result = **MBSCAN** (*string*, *set* [, *back*])

string
(Input) Character*(*). String to be searched for the presence of any character in *set*.

set
(Input) Character*(*). Characters to search for.

back
(Optional; input) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results:

The result type is INTEGER(4). If *back* is .FALSE. or is omitted, returns the position of the leftmost character in *string* that is in *set*. If *back* is .TRUE., returns the rightmost character in *string* that is in *set*. If no characters in *string* are in *set*, returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SCAN, MBINDEX, MBVERIFY

MBStrLead

NLS Function: Performs a context-senstive test to determine whether a given character byte in a string is a multibyte-character lead byte.

Module: USE DFNLS

Syntax

result = **MBStrLead** (*string*, *position*)

string
(Input) Character*(*). String containing the character byte to be tested for lead status.

position
(Input) INTEGER(4). Position in *string* of the character byte in the string to be tested.

Results:

The result type is LOGICAL(4). The result is .TRUE. if the character byte in *position* of *string* is a lead byte; otherwise, .FALSE..

MBStrLead is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context.

MBLead is passed only one character at a time and must start on a lead byte and step through a string one character at a time to establish context for the character. So, **MBStrLead** can be much slower than **MBLead** (up to *n* times slower, where *n* is the length of the string).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MBLead

MBVERIFY

NLS Function: Performs the same function as **VERIFY** except that the strings manipulated can contain multibyte characters.

Module: USE DFNLS

Syntax

result = **MBVERIFY** (*string*, *set* [, *back*])

string
(Input) Character*(*). String to be searched for presence of any character not in *set*.

set
(Input) Character*(*). Set of characters tested to verify that it includes all the characters in string.

back
(Optional; input) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results:

The result type is INTEGER(4). If *back* is .FALSE. or is omitted, returns the position of the leftmost character in *string* that is not in *set*. If *back* is .TRUE., returns the rightmost character in *string* that is not in *set*. If all the characters in *string* are in *set*, returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: VERIFY, MBINDEX, MBSCAN

MERGE

Elemental Intrinsic Function (Generic): Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.

Syntax

result = **MERGE** (*tsource, fsource, mask*)

tsource
(Input) Must be a scalar or array (of any data type).

fsource
(Input) Must be a scalar or array of the same type and type parameters as *tsource*.

mask
(Input) 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MVBITS

Examples

For MERGE (1.0, 0.0, R < 0), R = -3 has the value 1.0, and R = 7 has the value 0.0.

TSOURCE is the array

```
[ 1  3  5 ]
[ 2  4  6 ],
```

FSOURCE is the array

```
[ 8  9  0 ]
[ 1  2  3 ],
```

and MASK is the array

```
[ F  T  T]
[ T  T  F].
```

MERGE (TSOURCE, FSOURCE, MASK) produces the result:

```
[ 8  3  5 ]
[ 2  4  3 ].
```

The following shows another example:

```
INTEGER tsource(2, 3), fsource(2, 3), AR1 (2, 3)
LOGICAL mask(2, 3)
tsource = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2, 3/))
fsource = RESHAPE((/7, 0, 8, -1, 9, -2/), (/2, 3/))
mask = RESHAPE((/.TRUE., .FALSE., .FALSE., .TRUE.,          &
                .TRUE., .FALSE./), (/2,3/))
! tsource is  1 2 3 , fsource is  7  8  9 , mask is  T F T
!             4 5 6               0 -1 -2            F T F

AR1 = MERGE(tsource, fsource, mask) ! returns  1 8  3
                                    !          0 5 -2
END
```

MESSAGE

Compiler Directive: Specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.

Syntax

*c***DEC$ MESSAGE**:*string*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

string
Is a character constant specifying a message.

The following form is also allowed: `!MS$MESSAGE:string`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: General Compiler Directives.

Example

```
!DEC$ MESSAGE:'Compiling Sound Speed Equations'
```

MESSAGEBOXQQ

QuickWin Function: Displays a message box in a QuickWin window.

Module: USE DFLIB

Syntax

result = **MESSAGEBOXQQ** (*msg*, *caption*, *mtype*)

msg
(Input) Character*(*). Null-terminated C string. Message the box displays.

caption
(Input) Character*(*). Null-terminated C string. Caption that appears in the title bar.

mtype
(Input) INTEGER(4). Symbolic constant that determines the objects (buttons and icons) and attributes of the message box. You can combine several constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) using an inclusive OR (IOR or OR). The symbolic constants and their associated objects or attributes are:

- **MB$ABORTRETRYIGNORE**: The Abort, Retry, and Ignore buttons.
- **MB$DEFBUTTON1**: The first button is the default.
- **MB$DEFBUTTON2**: The second button is the default.
- **MB$DEFBUTTON3**: The third button is the default.
- **MB$ICONASTERISK**: Lowercase *i* in blue circle icon.
- **MB$ICONEXCLAMATION**: The exclamation-mark icon.
- **MB$ICONHAND**: The stop-sign icon.
- **MB$ICONINFORMATION**: Lowercase *i* in blue circle icon.
- **MB$ICONQUESTION**: The question-mark icon.
- **MB$ICONSTOP**: The stop-sign icon.
- **MB$OK**: The OK button.
- **MB$OKCANCEL**: The OK and Cancel buttons.
- **MB$RETRYCANCEL**: The Retry and Cancel buttons.
- **MB$SYSTEMMODAL**: Box is system-modal: all applications are suspended until the user responds.
- **MB$YESNO**: The Yes and No buttons.
- **MB$YESNOCANCEL**: The Yes, No, and Cancel buttons.

Results:

The result type is INTEGER(4). The result is zero if memory is not sufficient for displaying the message box. Otherwise, the result is one of the following values, indicating the user's response to the message box:

- **MB$IDABORT**: The Abort button was pressed.
- **MB$IDCANCEL**: The Cancel button was pressed.
- **MB$IDIGNORE**: The Ignore button was pressed.
- **MB$IDNO**: The No button was pressed.
- **MB$IDOK**: The OK button was pressed.
- **MB$IDRETRY**: The Retry button was pressed.
- **MB$IDYES**: The Yes button was pressed.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: ABOUTBOXQQ, SETMESSAGEQQ, Using QuickWin

Example

```
! Build as QuickWin app
USE DFLIB
message = MESSAGEBOXQQ('Do you want to continue?'C,    &
          'Matrix'C,   &
          MB$ICONQUESTION.OR.MB$YESNO.OR.MB$DEFBUTTON1)
END
```

MIN

Elemental Intrinsic Function (Generic): Returns the minimum value of the arguments.

Syntax

result = **MIN** (*a1*, *a2* [, *a3*...])

a1, *a2*, *a3*
(Input) 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 type is integer. For **AMIN0**, **AIMIN0**, **AJMIN0**, and **AKMIN0**, the result type is 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 [3, 4]	INTEGER(4)	REAL(4)
KMIN0 [5]	INTEGER(8)	INTEGER(8)
AKMIN0 [5]	INTEGER(8)	REAL(4)
IMIN1	REAL(4)	INTEGER(2)
MIN1 [4, 6, 7]	REAL(4)	INTEGER(4)
KMIN1 [5]	REAL(4)	INTEGER(8)
AMIN1 [8]	REAL(4)	REAL(4)
DMIN1	REAL(8)	REAL(8)
QMIN1 [9]	REAL(16)	REAL(16)

[1] These specific functions cannot be passed as actual arguments.
[2] Or JMIN0.
[3] Or AJMIN0. AMIN0 is the same as REAL (MIN).
[4] In Fortran 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.
[5] Alpha only
[6] Or JMIN1. MIN1 is the same as INT (MIN).
[7] The setting of compiler option /integer_size can affect MIN1.
[8] The setting of compiler option /real_size can affect AMIN1.
[7] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAX

Examples

MIN (2.0, -8.0, 6.0) has the value -8.0.

MIN (14, 32, -50) has the value -50.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MIN (5, 6, 7)           ! returns 5
m2 = MIN1 (-5.7, 1.23, -3.8) ! returns -5
```

```
r1 = AMIN0 (-5, -6, -7)      ! returns -7.0
r2 = AMIN1(-5.7, 1.23, -3.8) ! returns -5.7
```

MINEXPONENT

Inquiry Intrinsic Function (Generic): Returns the minimum exponent in the model representing the same type and kind parameters as the argument.

Syntax

result = **MINEXPONENT** (*x*)

x
(Input) 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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAXEXPONENT

Examples

If X is of type REAL(4), MINEXPONENT (X) has the value -125.

The following shows another example:

```
REAL(8) r1  ! DOUBLE PRECISION REAL
INTEGER i
i = MINEXPONENT (r1) ! returns - 1021.
```

MINLOC

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **MINLOC** (*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer or real.

dim

(Optional; input) 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
(Optional; input) Must be a logical array that is conformable with *array*.

Results:

The result is an array of type default integer.

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*.
- 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, the value of the result is undefined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAXLOC, MINVAL, MAXVAL

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

```
[  4    0   -3    2 ]
```

```
  [  3    1   -2    6 ]
  [ -1   -4    5   -5 ].
```

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.

The following shows another example:

```
INTEGER i, minl(1)
INTEGER array(2, 3)
INTEGER, ALLOCATABLE :: AR1(:)
! put values in array
array = RESHAPE((/-7, 1, -2, -9, 5, 0/),(/2, 3/))
!   array is   -7 -2 5
!               1 -9 0
i = SIZE(SHAPE(array))  ! Get the number of dimensions
                        ! in array
ALLOCATE (AR1 (i) )     ! Allocate AR1 to number
                        ! of dimensions in array
AR1 = MINLOC (array, MASK = array .GT. -5) ! Get the
                        ! location (subscripts) of
                        ! smallest element greater
                        ! than -5 in array

!
! MASK = array .GT. -5 creates a mask array the same
! size and shape as array whose elements are .TRUE. if
! the corresponding element in array is greater than
! -5, and .FALSE. if it is not. This mask causes MINLOC
! to return the index of the element in array with the
! smallest value greater than -5.
!
!array is  -7 -2 5 and MASK= array .GT. -5 is  F T T
!           1 -9 0                              T F T
! and AR1 = MINLOC(array, MASK = array .GT. -5) returns
! (1, 2), the location of the element with value -2

minl = MINLOC((/-7,2,-7,5/))   ! returns 1, first
                               ! occurrence of minimum
END
```

MINVAL

Transformational Intrinsic Function (Generic): 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.

Syntax

result = **MINVAL**(*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer or real.

dim
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.

mask
(Optional; input) 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 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MAXVAL, MINLOC, MAXLOC

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

```
[ 2  3  4 ]
[ 5  6  7 ].
```

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.

The following shows another example:

```
INTEGER array(2, 3), i(2), minv
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/), (/2, 3/))
!    array is    1 5 3
!                4 2 6
i = SHAPE(array)    ! i = [2 3]
ALLOCATE(AR1(i(2))) ! dimension AR1 to number of
                    ! elements in dimension 2
                    ! (a column) of array.
ALLOCATE(AR2(i(1))) ! dimension AR2 to number of
                    ! elements in dimension 1
                    ! (a row) of array
minv = MINVAL(array, MASK = array .GT. 4) ! returns 5
AR1 = MINVAL(array, DIM = 1)   ! returns [ 1 2 3 ]
AR2 = MINVAL(array, DIM = 2)   ! returns [ 1 2 ]
END
```

MOD

Elemental Intrinsic Function (Generic): Returns the remainder when the first argument is divided by the second argument.

Syntax

result = **MOD** (*a*, *p*)

a
(Input) Must be of type integer or real.

p
(Input)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 [2]	INTEGER(8)	INTEGER(8)
AMOD [3]	REAL(4)	REAL(4)
QMOD [4]	REAL(16)	REAL(16)

[1] Or JMOD.
[2] Alpha only
[3] The setting of compiler option /real_size can affect AMOD.
[4] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MODULO

Examples

MOD (7, 3) has the value 1.

MOD (9, -6) has the value 3.

MOD (-9, 6) has the value -3.

The following shows more examples:

```
INTEGER I
REAL R
R = MOD(9.0, 2.0)  ! returns 1.0
I = MOD(18, 5)     ! returns 3
I = MOD(-18, 5)    ! returns -3
```

MODIFYMENUFLAGSQQ

QuickWin Function: Modifies a menu item's state.

Module: USE DFLIB

Syntax

result = **MODIFYMENUFLAGSQQ** (*menuID, itemID, flag)*

menuID
(Input) INTEGER(4). Identifies the menu containing the item whose state is to be modified, starting with 1 as the leftmost menu.

itemID
(Input) INTEGER(4). Identifies the menu item whose state is to be modified, starting with 0 as the top item.

flags
(Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see below). The following constants are available:

- **$MENUGRAYED**: Disables and grays out the menu item.
- **$MENUDISABLED**: Disables but does not gray out the menu item.
- **$MENUENABLED**: Enables the menu item.
- **$MENUSEPARATOR**: Draws a separator bar.
- **$MENUCHECKED**: Puts a check by the menu item.
- **$MENUUNCHECKED**: Removes the check by the menu item.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: APPENDMENUQQ, DELETEMENUQQ, INSERTMENUQQ MODIFYMENUROUTINEQQ, MODIFYMENUSTRINGQQ, Using QuickWin

Example

```
USE DFLIB
LOGICAL(4)    result
CHARACTER(20) str

! Append item to the bottom of the first (FILE) menu
```

```
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Gray out and disable the first two menu items in the
! first (FILE) menu
result = MODIFYMENUFLAGSQQ (1, 1, $MENUGRAYED)
result = MODIFYMENUFLAGSQQ (1, 2, $MENUGRAYED)
END
```

MODIFYMENUROUTINEQQ

QuickWin Function: Changes a menu item's callback routine.

Module: USE DFLIB

Syntax

result = **MODIFYMENUROUTINEQQ** (*menuIdD, itemID, routine*)

menuID
(Input) INTEGER(4). Identifies the menu that contains the item whose callback routine is be changed, starting with 1 as the leftmost menu.

itemID
(Input) INTEGER(4). Identifies the menu item whose callback routine is to be changed, starting with 0 as the top item.

routine
(Input) EXTERNAL. Callback subroutine called if the menu item is selected. All routines must take a single LOGICAL parameter that indicates whether the menu item is checked or not. The following predefined routines are available for assigning to menus:

- **WINPRINT**: Prints the program.
- **WINSAVE**: Saves the program.
- **WINEXIT**: Terminates the program.
- **WINSELTEXT**: Selects text from the current window.
- **WINSELGRAPH**: Selects graphics from the current window.
- **WINSELALL**: Selects the entire contents of the current window.
- **WINCOPY**: Copies the selected text and/or graphics from the current window to the Clipboard.
- **WINPASTE**: Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a **READ.**
- **WINCLEARPASTE**: Clears the paste buffer.
- **WINSIZETOFIT**: Sizes output to fit window.
- **WINFULLSCREEN**: Displays output in full screen.
- **WINSTATE**: Toggles between pause and resume states of text output.
- **WINCASCADE**: Cascades active windows.
- **WINTILE**: Tiles active windows.
- **WINARRANGE**: Arranges icons.
- **WINSTATUS**: Enables a status bar.
- **WININDEX**: Displays the index for QuickWin Help.

- **WINUSING**: Displays information on how to use Help.
- **WINABOUT**: Displays information about the current QuickWin application.
- **NUL**: No callback routine.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Compatibility

QUICKWIN GRAPHICS LIB

See Also: APPENDMENUQQ, DELETEMENUQQ, INSERTMENUQQ, MODIFYMENUFLAGSQQ, MODIFYMENUSTRINGQQ, Using QuickWin

MODIFYMENUSTRINGQQ

QuickWin Function: Changes a menu item's text string.

Module: USE DFLIB

Syntax

result = **MODIFYMENUSTRINGQQ** (*menuID*, *itemID*, *text*)

menuID
(Input) INTEGER(4). Identifies the menu containing the item whose text string is to be changed, starting with 1 as the leftmost item.

itemID
(Input) INTEGER(4). Identifies the menu item whose text string is to be changed, starting with 0 as the top menu item.

text
(Input) Character*(*). Menu item name. Must be a null-terminated C string. For example, words of text'C.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise,.FALSE..

You can add access keys in your text strings by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, use "P&rint"C as *text*.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: APPENDMENUQQ, DELETEMENUQQ, INSERTMENUQQ, SETMESSAGEQQ MODIFYMENUFLAGSQQ, MODIFYMENUROUTINEQQ, Using QuickWin

Example

```
USE DFLIB
LOGICAL(4) result
CHARACTER(25) str

! Append item to the bottom of the first (FILE) menu
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Change the name of the first item in the first menu
str ='&Browse'C
result = MODIFYMENUSTRINGQQ (1, 1, str)
END
```

MODULE

Statement: Marks the beginning of a module program unit, which contains specifications and definitions that can be made accessible to other program units.

Syntax

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 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).

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PUBLIC, PRIVATE, USE, Procedure Interfaces, Program Units and Procedures

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 RESULTS
...
CONTAINS
  FUNCTION MOD_RESULTS(X,Y)  ! A module procedure
  ...
```

```
  END FUNCTION MOD_RESULTS
END MODULE RESULTS
```

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.

The following shows another example:

```
 MODULE mod1
   REAL(8) a,b,c,d
   INTEGER(4) Int1, Int2, Int3
 CONTAINS
   function fun1(x)
 ....
   end function fun1
 END MODULE
```

MODULE PROCEDURE

Statement: Identifies module procedures in an interface block that specifies a generic name. For more information, see INTERFACE and MODULE.

Example

```
!A program that changes non-default integers and reals !into default integers and r
   PROGRAM CHANGE_KIND
   USE Module1
   INTERFACE DEFAULT
     MODULE PROCEDURE Sub1, Sub2
   END INTERFACE

   integer(2) in
   integer indef
   indef = DEFAULT(in)
   END PROGRAM
! procedures sub1 and sub2 defined as follows:
    MODULE Module1
    CONTAINS
      FUNCTION Sub1(y)
        REAL(8) y
        sub1 = REAL(y)
      END FUNCTION
      FUNCTION Sub2(z)
        INTEGER(2) z
        sub2 = INT(z)
      END FUNCTION
    END MODULE
```

MODULO

Elemental Intrinsic Function (Generic): Returns the modulo of the arguments.

Syntax

result = **MODULO** (*a*, *p*)

a
(Input) Must be of type integer or real.

p
(Input) Must have the same type and kind parameters as *a*.

Results:

The result type is the same *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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MOD

Examples

MODULO (7, 3) has the value 1.

MODULO (9, -6) has the value -3.

MODULO (-9, 6) has the value 3.

The following shows more examples:

```
INTEGER I
REAL R
I= MODULO(8, 5)         ! returns 3          Note: q=1
I= MODULO(-8, 5)        ! returns 2          Note: q=-2
I= MODULO(8, -5)        ! returns -2         Note: q=-2
R= MODULO(7.285, 2.35)  ! returns 0.2350001  Note: q=3
R= MODULO(7.285, -2.35) ! returns -2.115     Note: q=-4
```

MOVETO, MOVETO_W

Graphics Subroutine: Moves the current graphics position to a specified point. No drawing occurs.

Module: USE DFLIB

Syntax

CALL MOVETO (*x*, *y*, *t*)
CALL MOVETO_W (*wx*, *wy*, *w*)

x, *y*
(Input) INTEGER(2). Viewport coordinates of the new graphics position.

wx, *wy*
(Input) REAL(8). Window coordinates of the new graphics position.

t
(Output) Derived type xycoord. Viewport coordinates of the previous graphics position. The derived type xycoord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE xycoord
  INTEGER(2) xcoord  ! x coordinate
  INTEGER(2) ycoord  ! y coordinate
END TYPE xycoord
```

wt
(Output) Derived type wxycoord. Window coordinates of the previous graphics position. The derived type wxycoord is defined in DFLIB.F90 as follows:

```
TYPE wxycoord
  REAL(8) wx  ! x window coordinate
  REAL(8) wy  ! y window coordinate
END TYPE wxycoord
```

MOVETO sets the current graphics position to the viewport coordinate (*x*, *y*). **MOVETO_W** sets the current graphics position to the window coordinate (*wx*, *wy*).

MOVETO and **MOVETO_W** assign the coordinates of the previous position to *t* and *wt*, respectively.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCURRENTPOSITION, LINETO, OUTGTEXT

Example

```
! Build as QuickWin or Standard Graphics ap.
USE DFLIB
INTEGER(2) status, x, y
INTEGER(4) result
TYPE (xycoord) xy
```

```
RESULT = SETCOLORRGB(#FF0000) ! blue
x = 60
! Draw a series of lines
DO y = 50, 92, 3
  CALL MOVETO(x, y, xy)
  status = LINETO(INT2(x + 20), y)
END DO
END
```

MULT_HIGH (Alpha only)

Elemental Intrinsic Function (Specific): Multiplies two 64-bit unsigned integers.

Syntax

result = **MULT_HIGH** (*i*, *j*)

i
INTEGER(8).

j
INTEGER(8).

Results:

The result type is INTEGER(8). The result value is the upper (leftmost) 64 bits of the 128-bit unsigned result.

Example

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
```

MVBITS

Elemental Intrinsic Subroutine: Copies a sequence of bits (a bit field) from one location to another.

Syntax

CALL MVBITS (*from*, *frompos*, *len*, *to*, *topos*)

from
(Input) Integer. Can be of any integer type. It represents the location from which a bit field is transferred.

frompos
(Input) 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*).

len
(Input) Can be of any integer type; it must not be negative. It identifies the length of the field transferred from *from*.

to
(Input; output) 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.

topos
(Input) 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*).

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

You can also use the following specific routines:

IMVBITS	All arguments must be INTEGER(2)
JMVBITS	Arguments can be INTEGER(2) or INTEGER(4); at least one must be INTEGER (4)
KMVBITS [1]	Arguments can be INTEGER(2), INTEGER(4), or INTEGER(8); at least one must be INTEGER(8)

[1] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BIT_SIZE, IBCLR, IBSET, ISHFT, ISHFTC

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.

The following shows another example:

```
INTEGER(1) :: from = 13  ! 00001101
INTEGER(1) :: to = 6     ! 00000110
CALL MVBITS(from, 2, 2, to, 0) ! returns to = 00000111
END
```

NAMELIST

Statement: Associates a name with a list of variables. This group name can be referenced in some input/output operations.

Syntax

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:

- An array dummy argument with nonconstant bounds
- A variable with assumed character length
- An allocatable array

- An automatic object
- A pointer
- A variable of a type that has a pointer as an ultimate component
- A 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: READ, WRITE, Namelist Specifier, Namelist Input, Namelist Output.

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.

The following shows another example:

```
 NAMELIST /example/ i1, l1, r4, r8, z8, z16, c1, c10, iarray

 ! The corresponding input statements could be:
 &example
 i1  = 11
 l1  = .TRUE.
 r4  = 24.0
 r8  = 28.0d0
 z8  = (38.0, 0.0)
```

```
z16 = (316.0d0, 0.0d0)
c1  = 'A'
c10 = 'abcdefghij'
iarray(8) = 41, 42, 43
/
```

A sample program, NAMELIST.F90, is included in the \DF\SAMPLES\TUTORIAL subdirectory.

NARGS

Run-Time Function: Returns the total number of command-line arguments, including the command.

Module: USE DFLIB

Syntax

result = **NARGS** ( )

Results:

The result type is INTEGER(4). The result is the number of command-line arguments, including the command. For example, **NARGS** returns 4 for the command-line invocation of `PROG1 -g -c -a`.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETARG, IARGC

Example

```
    USE DFLIB
    INTEGER(2) result
    result = RUNQQ('myprog', '-c -r')
    END

!  MYPROG.F90 responds to command switches -r, -c,
!  and/or -d
USE DFLIB
INTEGER(4) count, num
INTEGER(2) i, status
CHARACTER(80) buf
REAL r1 / 0.0 /
COMPLEX c1 / (0.0,0.0) /
REAL(8) d1 / 0.0 /

num = 5
count = NARGS( )
DO i = 1, count-1
   CALL GETARG(i, buf, status)
   IF (buf(2:status) .EQ.'r') THEN
     r1 = REAL(num)
     WRITE (*,*) 'r1 = ',r1
   ELSE IF (buf(2:status) .EQ.'c') THEN
     c1 = CMPLX(num)
```

```
      WRITE (*,*) 'c1 = ', c1
    ELSE IF (buf(2:status) .EQ.'d') THEN
      d1 = DBLE(num)
      WRITE (*,*) 'd1 = ', d1
    ELSE
      WRITE(*,*) 'Invalid command switch'
      EXIT
    END IF
 END DO
 END
```

NEAREST

Elemental Intrinsic Function (Generic): Returns the nearest different number (representable on the processor) in a given direction.

Syntax

result = **NEAREST** (*x*, *s*)

x
(Input) Must be of type real.

s
(Input) Must be of type real and nonzero.

Results:

The result type is the same as *x*. The result has a value equal to the machine representable number that is different from and nearest to *x*, in the direction of infinity, with the same sign as *s*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EPSILON

Examples

If 3.0 and 2.0 are REAL(4) values, NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$, which equals approximately 3.0000002. (For more information on the model for REAL(4), see Model for Real Data.

The following shows another example:

```
 REAL(4) r1
 REAL(8) r2, result
 r1 = 3.0
 result = NEAREST (r1, -2.0)
 WRITE(*,*) result          ! writes 2.999999761581421

 ! When finding nearest to REAL(8), can't see
 ! the difference unless output in HEX
```

```
r2 = 111502.07D0
result = NEAREST(r2, 2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851ED
result = NEAREST(r2, -2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851EB
END
```

NINT

Elemental Intrinsic Function (Generic): Returns the nearest integer to the argument.

Syntax

result = **NINT** (*a* [, *kind*])

a
(Input) Must be of type real.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is integer. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, see the following table for the kind parameter. 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 [1, 2]	REAL(4)	INTEGER(4)
KNINT [3]	REAL(4)	INTEGER(8)
IIDNNT	REAL(8)	INTEGER(2)
IDNINT [2, 4]	REAL(8)	INTEGER(4)
KIDNNT [3]	REAL(8)	INTEGER(8)
IIQNNT [5]	REAL(16)	INTEGER(2)
IQNINT [2, 5]	REAL(16)	INTEGER(4)
KIQNNT [5, 6]	REAL(16)	INTEGER(8)

[1] Or JNINT.
[2] The setting of compiler option /integer_size can affect NINT, IDNINT, and IQNINT.
[3] Alpha only

[4] Or JIDNNT. For compatibility with older versions of Fortran, IDNINT can also be specified as a generic function.
[5] VMS, U*X
[6] This specific function cannot be passed as an actual argument.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ANINT, INT

Examples

NINT (3.879) has the value 4.

NINT (-2.789) has the value -3.

The following shows another example:

```
INTEGER(4) i1, i2
i1 = NINT(2.783) ! returns 3
i2 = IDNINT(-2.783D0) ! returns -3
```

NLSEnumCodepages

NLS Function: Returns an array containing the codepages supported by the system, with each array element describing one valid codepage.

Module: USE DFNLS

Syntax

ptr => **NLSEnumCodepages** ( )

Results:

The result is a pointer to an array of codepages, with each element describing one supported codepage.

After use, the pointer returned by **NLSEnumCodepages** should be deallocated with the **DEALLOCATE** statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSEnumLocales

NLSEnumLocales

NLS Function: Returns an array containing the language and country combinations supported by the system, in which each array element describes one valid combination.

Module: USE DFNLS

Syntax

ptr => **NLSEnumLocales** ( )

Results:

The result is a pointer to an array of locales, in which each array element describes one supported language and country combination. Each element has the following structure:

```
TYPE NLS$EnumLocale
  CHARACTER*(NLS$MaxLanguageLen)  Language
  CHARACTER*(NLS$MaxCountryLen)   Country
  INTEGER(4)                      DefaultWindowsCodepage
  INTEGER(4)                      DefaultConsoleCodepage
END TYPE
```

If the application is a Windows or QuickWin application, NLS$DefaultWindowsCodepage is the codepage used by default for the given language and country combination. If the application is a console application, NLS$DefaultConsoleCodepage is the codepage used by default for the given language and country combination.

Note: After use, the pointer returned by **NLSEnumLocales** should be deallocated with the DEALLOCATE statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSEnumCodepages

NLSFormatCurrency

NLS Function: Returns a correctly formatted currency string for the current locale.

Module: USE DFNLS

Syntax

result = **NLSFormatCurrency** (*outstr*, *instr* [, *flags*])

outstr
(Output) Character*(*). String containing the correctly formatted currency for the current locale. If *outstr* is longer than the formatted currency, it is blank-padded.

intstr
(Input) Character*(*). Number string to be formatted. Can contain only the characters 0’ through 9’, one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.

flags
(Optional; input) INTEGER(4). If specified, modifies the currency conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in DFNLS.F90) are:

- **NLS$Normal**: No special formatting
- **NLS$NoUserOverride**: Do not use user overrides

Results:

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- **NLS$ErrorInsufficentBuffer**: *outstr* buffer is too small
- **NLS$ErrorInvalidFlags**: *flags* has an illegal value
- **NLS$ErrorInvalidInput**: *instr* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSFormatNumber, NLSFormatDate, NLSFormatTime

Example

```
USE DFNLS
CHARACTER(40) str
INTEGER(4) i
i = NLSFormatCurrency(str, "1.23")
print *, str                                ! prints $1.23
i = NLSFormatCurrency(str, "1000000.99")
print *, str                                ! prints $1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatCurrency(str, "1.23")
print *, str                                ! prints 1 Pts
i = NLSFormatCurrency(str, "1000000.99")
print *, str                                ! prints 1.000.001 Pts
```

NLSFormatDate

NLS Function: Returns a correctly formatted string containing the date for the current locale.

Module: USE DFNLS

Syntax

result = **NLSFormatDate** (*outstr* [, *intime*] [, *flags*])

outstr
(Output) Character*(*). String containing the correctly formatted date for the current locale. If *outstr* is longer than the formatted date, it is blank-padded.

intime
(Optional; input) INTEGER(4). If specified, date to be formatted for the current locale. Must be an integer date such as the packed time created with **PACKTIMEQQ**. If you omit *intime*, the current system date is formatted and returned in *outstr*.

flags
(Optional; input) INTEGER(4). If specified, modifies the date conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in DFNLS.F90 in /DF/INCLUDE) are:

- **NLS$Normal**: No special formatting
- **NLS$NoUserOverride**: Do not use user overrides
- **NLS$UseAltCalendar**: Use the locale's alternate calendar
- **NLS$LongDate**: Use local long date format
- **NLS$ShortDate**: Use local short date format

Results:

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- **NLS$ErrorInsufficentBuffer**: *outstr* buffer is too small
- **NLS$ErrorInvalidFlags**: *flags* has an illegal value
- **NLS$ErrorInvalidInput**: *intime* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSFormatTime, NLSFormatCurrency, NLSFormatNumber

Examples

```
USE DFNLS
INTEGER(4) i
CHARACTER(40) str
i = NLSFORMATDATE(str, NLS$NORMAL)           ! 8/1/94
i = NLSFORMATDATE(str, NLS$USEALTCALENDAR)   ! 8/1/94
i = NLSFORMATDATE(str, NLS$LONGDATE)         ! Monday, August 1, 1994
i = NLSFORMATDATE(str, NLS$SHORTDATE)        ! 8/1/94
END
```

NLSFormatNumber

NLS Function: Returns a correctly formatted number string for the current locale.

Module: USE DFNLS

Syntax

result = **NLSFormatNumber** (*outstr*, *instr* [, *flags*])

outstr
(Output) Character*(*). String containing the correctly formatted number for the current locale. If *outstr* is longer than the formatted number, it is blank-padded.

instr
(Input) Character*(*). Number string to be formatted. Can only contain the characters 0' through 9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.

flags
(Optional; input) INTEGER(4). If specified, modifies the number conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in DFNLS.F90 in /DF/INCLUDE) are:

- **NLS$Normal**: No special formatting
- **NLS$NoUserOverride**: Do not use user overrides

Results:

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- **NLS$ErrorInsufficentBuffer**: *outstr* buffer is too small
- **NLS$ErrorInvalidFlags**: *flags* has an illegal value
- **NLS$ErrorInvalidInput**: *instr* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSFormatTime, NLSFormatCurrency, NLSFormatDate

Example

```
USE DFNLS
CHARACTER(40) str
INTEGER(4) i
```

```
i = NLSFormatNumber(str, "1.23")
print *, str                              ! prints 1.23
i = NLSFormatNumber(str, "1000000.99")
print *, str                              ! prints 1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatNumber(str, "1.23")
print *, str                              ! prints 1,23
i = NLSFormatNumber(str, "1000000.99")
print *, str                              ! prints 1.000.000,99
```

NLSFormatTime

NLS Function: Returns a correctly formatted string containing the time for the current locale.

Module: USE DFNLS

Syntax

result = **NLSFormatTime** (*outstr* [, *intime*] [, *flags*])

outstr
(Output) Character*(*). String containing the correctly formatted time for the current locale. If *outstr* is longer than the formatted time, it is blank-padded.

intime
(Optional; input) INTEGER(4). If specified, time to be formatted for the current locale. Must be an integer time such as the packed time created with **PACKTIMEQQ**. If you omit *intime*, the current system time is formatted and returned in *outstr*.

flags
(Optional; input) INTEGER(4). If specified, modifies the time conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in DFNLS.F90 in /DF/INCLUDE) are:

- **NLS$Normal**: No special formatting
- **NLS$NoUserOverride**: Do not use user overrides
- **NLS$NoMinutesOrSeconds**: Do not return minutes or seconds
- **NLS$NoSeconds**: Do not return seconds
- **NLS$NoTimeMarker**: Do not add a time marker string
- **NLS$Force24HourFormat**: Return string in 24 hour format

Results:

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- **NLS$ErrorInsufficentBuffer**: *outstr* buffer is too small
- **NLS$ErrorInvalidFlags**: *flags* has an illegal value
- **NLS$ErrorInvalidInput**: *intime* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSFormatCurrency, NLSFormatDate, NLSFormatNumber

Examples

```
USE DFNLS
INTEGER(4) i
CHARACTER(20) str
i = NLSFORMATTIME(str, NLS$NORMAL)              ! 11:38:28 PM
i = NLSFORMATTIME(str, NLS$NOMINUTESORSECONDS)  ! 11 PM
i = NLSFORMATTIME(str, NLS$NOTIMEMARKER)        ! 11:38:28 PM
i = NLSFORMATTIME(str, IOR(NLS$FORCE24HOURFORMAT,                &
&                      NLS$NOSECONDS))          ! 23:38 PM
END
```

NLSGetEnvironmentCodepage

NLS Function: Returns the codepage number for the system (Window) codepage or the console codepage.

Module: USE DFNLS

Syntax

result = **NLSGetEnvironmentCodepage** (*flags*)

flags
(Input) INTEGER(4). Tells the function which codepage number to return. Available values (defined in DFNLS.F90 in /DF/INCLUDE) are:

- **NLS$ConsoleEnvironmentCodepage**: Gets the codepage for the console
- **NLS$WindowsEnvironmentCodepage**: Gets the current Windows codepage

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, it returns one of the following error codes:

- **NLS$ErrorInvalidFlags**: *flags* has an illegal value
- **NLS$ErrorNoConsole**: There is no console associated with the given application; therefore, operations with the console codepage are not possible

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSSetEnvironmentCodepage

NLSGetLocale

NLS Subroutine: Retrieves the current language, country, or codepage.

Module: USE DFNLS

Syntax

CALL NLSGetLocale ([*language*] [, *country*] [, *codepage*])

language
(Optional; output) Character*(*). Current language.

country
(Optional; output) Character*(*). Current country.

codepage
(Optional; output) INTEGER(4). Current codepage.

NLSGetLocale returns a valid codepage in *codepage*. It does not return one of the **NLS$**... symbolic constants that can be used with **NLSSetLocale**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSSetLocale

Example

```
USE DFNLS
CHARACTER(50) cntry, lang
INTEGER(4)    code
CALL NLSGetLocale (lang, cntry, code)    ! get all three
CALL NLSGetLocale (CODEPAGE = code)      ! get the codepage
CALL NLSGetLocale (COUNTRY = cntry, CODEPAGE =code) ! get country
                                                    ! and codepage
```

NLSGetLocaleInfo

NLS Function: Returns information about the current locale.

Module: USE DFNLS

Syntax

result = **NLSGetLocaleInfo** (*type*, *outstr*)

type
(Input) INTEGER(4). NLS parameter requested. A list of parameter names is given in the NLS Locale Info Parameters Table.

outstr
(Output) Character*(*). Parameter setting for the current locale. All parameter settings placed in *outstr* are character strings, even numbers. If a parameter setting is numeric, the ASCII representation of the number is used. If the requested parameter is a date or time string, an explanation of how to interpret the format in *outstr* is given in NLS Date and Time Format.

Results:

The result type is INTEGER(4). The result is the number of characters written to *outstr* if successful, or if *outstr* has 0 length, the number of characters required to hold the requested information. Otherwise, the result is one of the following error codes (defined in DFNLS.F90):

- **NLS$ErrorInvalidLIType**: The given *type* is invalid
- **NLS$ErrorInsufficientBuffer**: The *outstr* buffer was too small, but was not 0 (so that the needed size would be returned)

The NLS$LI parameters are used for the argument *type* and select the locale information returned by **NLSGetLocaleInfo** in *outstr*. You can perform an inclusive OR with NLS$NoUserOverride and any NLS$LI parameter. This causes **NLSGetLocaleInfo** to bypass any user overrides and always return the system default value. The following table lists the NLS$LI parameters and describes each.

NLS Locale Info Parameters Table	
Parameter	**Description**
NLS$LI_ILANGUAGE	An ID indicating the language.
NLS$LI_SLANGUAGE	The full localized name of the language.
NLS$LI_SENGLANGUAGE	The full English name of the language from the ISO Standard 639. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SABBREVLANGNAME	The abbreviated name of the language, created by taking the 2-letter language abbreviation as found in ISO Standard 639 and adding a third letter as appropriate to indicate the sublanguage.
NLS$LI_SNATIVELANGNAME	The native name of the language.
NLS$LI_ICOUNTRY	The country code, based on international phone codes, also referred to as IBM country codes.

NLS$LI_SCOUNTRY	The full localized name of the country.
NLS$LI_SENGCOUNTRY	The full English name of the country. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SABBREVCTRYNAME	The abbreviated name of the country as per ISO Standard 3166.
NLS$LI_SNATIVECTRYNAME	The native name of the country.
NLS$LI_IDEFAULTLANGUAGE	Language ID for the principal language spoken in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTCOUNTRY	Country code for the principal country in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTANSICODEPAGE	ANSI code page associated with this locale.
NLS$LI_IDEFAULTOEMCODEPAGE	OEM code page associated with the locale.
NLS$LI_SLIST	Character(s) used to separate list items, for example, comma in many locales.
NLS$LI_IMEASURE	This value is 0 if the metric system (S.I.) is used and 1 for the U.S. system of measurements.
NLS$LI_SDECIMAL	The character(s) used as decimal separator. This is restricted such that it can not be set to digits 0 - 9.
NLS$LI_STHOUSAND	The character(s) used as separator between groups of digits left of the decimal. This is restricted such that it can not be set to digits 0 - 9.
NLS$LI_SGROUPING	Sizes for each group of digits to the left of the decimal. An explicit size is needed for each group; sizes are separated by semicolons. If the last value is 0 the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_IDIGITS	The number of decimal digits.
NLS$LI_ILZERO	Determines whether to use leading zeros in decimal fields: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_INEGNUMBER	Determines how negative numbers are represented: 0 - Puts negative numbers in parentheses: (1.1) 1 - Puts a minus sign in front: -1.1

	2 - Puts a minus sign followed by a space in front: -1.1 3 - Puts a minus sign after: 1.1- 4 - Puts a space then a minus sign after: 1.1 -
NLS$LI_SNATIVEDIGITS	The ten characters that are the native equivalent to the ASCII 0-9.
NLS$LI_SCURRENCY	The string used as the local monetary symbol. Cannot be set to digits 0-9.
NLS$LI_SINTLSYMBOL	Three characters of the International monetary symbol specified in ISO 4217 "Codes for the Representation of Currencies and Funds", followed by the character separating this string from the amount.
NLS$LI_SMONDECIMALSEP	The character(s) used as monetary decimal separator. This is restricted such that it cannot be set to digits 0-9.
NLS$LI_SMONTHOUSANDSEP	The character(s) used as monetary separator between groups of digits left of the decimal. Cannot be set to digits 0-9.
NLS$LI_SMONGROUPING	Sizes for each group of monetary digits to the left of the decimal. If the last value is 0, the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_ICURRDIGITS	Number of decimal digits for the local monetary format.
NLS$LI_IINTLCURRDIGITS	Number of decimal digits for the international monetary format.
NLS$LI_ICURRENCY	Determines how positive currency is represented: 0 - Puts currency symbol in front with no separation: $1.1 1 - Puts currency symbol in back with no separation: 1.1$ 2 - Puts currency symbol in front with single space after: $ 1.1 3 - Puts currency symbol in back with single space before: 1.1 $
NLS$LI_INEGCURR	Determines how negative currency is represented: 0 ($1.1) 1 -$1.1 2 $-1.1 3 $1.1- 4 (1.1$) 5 -1.1$

	6 1.1-$ 7 1.1$- 8 -1.1 $ (space before $) 9 -$ 1.1 (space after $) 10 1.1 $- (space before $) 11 $ 1.1- (space after $) 12 $ -1.1 (space after $) 13 1.1- $ (space before $) 14 ($ 1.1) (space after $) 15 (1.1 $) (space before $)
NLS$LI_SPOSITIVESIGN	String value for the positive sign. Cannot be set to digits 0-9.
NLS$LI_SNEGATIVESIGN	String value for the negative sign. Cannot be set to digits 0-9.
NLS$LI_IPOSSIGNPOSN	Determines the formatting index for positive values: 0 - Parenthesis surround the amount and the monetary symbol 1 - The sign string precedes the amount and the monetary symbol 2 - The sign string follows the amount and the monetary symbol 3 - The sign string immediately precedes the monetary symbol 4 - The sign string immediately follows the monetary symbol
NLS$LI_INEGSIGNPOSN	Determines the formatting index for negative values. Same values as for NLS$LI_IPOSSIGNPOSN.
NLS$LI_IPOSSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a positive amount.
NLS$LI_IPOSSEPBYSPACE	1 if the monetary symbol is separated by a space from a positive amount, 0 otherwise.
NLS$LI_INEGSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a negative amount.
NLS$LI_INEGSEPBYSPACE	1 if the monetary symbol is separated by a space from a negative amount, 0 otherwise.
NLS$LI_STIMEFORMAT	Time formatting string. See the NLS Date and Time Format section for explanations of the valid strings.
NLS$LI_STIME	Character(s) for the time separator. Cannot be set to digits 0-9.
NLS$LI_ITIME	Time format: 0 - Use 12-hour format

	1 - Use 24-hour format
NLS$LI_ITLZERO	Determines whether to use leading zeros in time fields: 0 - Use no leading zeros 1 - Use leading zeros for hours
NLS$LI_S1159	String for the AM designator.
NLS$LI_S2359	String for the PM designator.
NLS$LI_SSHORTDATE	Short Date formatting string for this locale. The d, M and y should have the day, month, and year substituted, respectively. See the NLS Date and Time Format section for explanations of the valid strings.
NLS$LI_SDATE	Character(s) for the date separator. Cannot be set to digits 0-9.
NLS$LI_IDATE	Short Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_ICENTURY	Specifies whether to use full 4-digit century for the short date only: 0 - Two-digit year 1 - Full century
NLS$LI_IDAYLZERO	Specifies whether to use leading zeros in day fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_IMONLZERO	Specifies whether to use leading zeros in month fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_SLONGDATE	Long Date formatting string for this locale. The string returned may contain a string within single quotes (’ ’). Any characters within single quotes should be left as is. The d, M and y should have the day, month, and year substituted, respectively.
NLS$LI_ILDATE	Long Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_ICALENDARTYPE	Specifies which type of calendar is currently being used:

	1 - Gregorian (as in United States) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)
NLS$LI_IOPTIONALCALENDAR	Specifies which additional calendar types are valid and available for this locale. This can be a null separated list of all valid optional calendars: 0 - No additional types valid 1 - Gregorian (localized) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)
NLS$LI_IFIRSTDAYOFWEEK	Specifies which day is considered first in a week: 0 - SDAYNAME1 1 - SDAYNAME2 2 - SDAYNAME3 3 - SDAYNAME4 4 - SDAYNAME5 5 - SDAYNAME6 6 - SDAYNAME7
NLS$LI_IFIRSTWEEKOFYEAR	Specifies which week of the year is considered first: 0 - Week containing 1/1 1 - First full week following 1/1 2 - First week containing at least 4 days
NLS$LI_SDAYNAME1 - NLS$LI_SDAYNAME7	Native name for each day of the week. 1= Monday, 2 = Tuesday, etc.
NLS$LI_SABBREVDAYNAME1 - NLS$LI_SABBREVDAYNAME7	Native abbreviated name for each day of the week. 1 = Mon, 2 = Tue, etc.
NLS$LI_SMONTHNAME1 - NLS$LI_SMONTHNAME13	Native name for each month. 1 = January, 2 = February, etc. 13 = the 13th month, if it exists in the locale.
NLS$LI_SABBREVMONTHNAME1 - NLS$LI_SABBREVMONTHNAME13	Native abbreviated name for each month. 1 = Jan, 2 = Feb, etc. 13 = the 13th month, if it exists in the locale.

NLS Date and Time Format

When **NLSGetLocaleInfo** (*type*, *outstr*) returns information about the date and time formats of the current locale, the value returned in *outstr* can be interpreted according to the following tables. Any text returned within a date and time string that is enclosed within single quotes should be left in the string in its exact form; that is, do not change the text or the location within the string.

Day

The day can be displayed in one of four formats using the letter "d". The table below shows the four variations:

d	Day of the month as digits without leading zeros for single-digit days
dd	Day of the month as digits with leading zeros for single-digit days
ddd	Day of the week as a three-letter abbreviation (SABBREVDAYNAME)
dddd	Day of the week as its full name (SDAYNAME)

Month

The month can be displayed in one of four formats using the letter "M". The uppercase "M" distinguishes months from minutes. The table below shows the four variations:

M	Month as digits without leading zeros for single-digit months
MM	Month as digits with leading zeros for single-digit months
MMM	Month as a three-letter abbreviation (SABBREVMONTHNAME)
MMMM	Month as its full name (SMONTHNAME)

Year

The year can be displayed in one of three formats using the letter "y". The table below shows the three variations:

y	Year represented by only the last digit
yy	Year represented by only the last two digits
yyyy	Year represented by the full 4 digits

Period/Era

The period/era string is displayed in a single format using the letters "gg".

gg	Period/Era string

Time

The time can be displayed in one of many formats using the letter "h" or "H" to denote hours, the letter "m" to denote minutes, the letter "s" to denote seconds and the letter "t" to denote the time marker. The table below shows the numerous variations of the time format. Lowercase "h" denotes the 12 hour clock and uppercase "H" denotes the 24 hour clock. The lowercase "m" distinguishes minutes from months.

h	Hours without leading zeros for single-digit hours (12 hour clock)
hh	Hours with leading zeros for single-digit hours (12 hour clock)
H	Hours without leading zeros for single-digit hours (24 hour clock)
HH	Hours with leading zeros for single-digit hours (24 hour clock)
m	Minutes without leading zeros for single-digit minutes
mm	Minutes with leading zeros for single-digit minutes
s	Seconds without leading zeros for single-digit seconds
ss	Seconds with leading zeros for single-digit seconds
t	One-character time marker string
tt	Multicharacter time marker string

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSGetLocale, NLSFormatDate, NLSFormatTime, NLSSetLocale

Example

```
USE DFNLS
INTEGER(4) strlen
CHARACTER(40) str
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str    ! prints Monday if language is English
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str    ! prints Tuesday if language is English
```

NLSSetEnvironmentCodepage

NLS Function: Sets the codepage for the current console. The specified codepage affects the current console program and any other programs launched from the same console. It does not affect other open consoles or any consoles opened later.

Module: USE DFNLS

Syntax

result = **NLSSetEnvironmentCodepage** (*codepage*, *flags*)

codepage
(Input) INTEGER(4). Number of the codepage to set as the console codepage.

flags
(Input) INTEGER(4). Must be set to NLS$ConsoleEnvironmentCodepage.

Results:

The result type is INTEGER(4). The result is zero if successful. Otherwise, returns one of the following error codes (defined in DFNLS.F90 in /DF/INCLUDE):

- **NLS$ErrorInvalidCodepage**: *codepage* is invalid or not installed on the system
- **NLS$ErrorInvalidFlags**: *flags* is not valid
- **NLS$ErrorNoConsole**: There is no console associated with the given application; therefore operations, with the console codepage are not possible

The *flags* argument must be NLS$ConsoleEnvironmentCodepage; it cannot be NLS$WindowsEnvironmentCodepage. **NLSSetEnvironmentCodepage** does not affect the Windows codepage.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSGetEnvironmentCodepage

NLSSetLocale

NLS Function: Sets the current language, country, or codepage.

Module: USE DFNLS

Syntax

result = **NLSSetLocale** (*language* [, *country*] [, *codepage*])

language
(Input) Character*(*). One of the languages supported by the Win32 NLS APIs.

country
(Optional; input) Character*(*). If specified, characterizes the language further. If omitted, the default country for the language is set.

codepage
(Optional; input) INTEGER(4). If specified, codepage to use for all character-oriented NLS functions. Can be any valid supported codepage or one of the following predefined values (defined in DFNLS.F90 in /DF/INCLUDE/):

- **NLS$CurrentCodepage**: The codepage is not changed. Only the language and country settings are altered by the function.
- **NLS$ConsoleEnvironmentCodepage**: The codepage is changed to the default environment codepage currently in effect for console programs.
- **NLS$ConsoleLanguageCodepage**: The codepage is changed to the default console codepage for the language and country combination specified.
- **NLS$WindowsEnvironmentCodepage**: The codepage is changed to the default environment codepage currently in effect for Windows programs.
- **NLS$WindowsLanguageCodepage**: The codepage is changed to the default Windows codepage for the language and country combination specified.

If you omit *codepage*, it defaults to NLS$WindowsLanguageCodepage. At program startup, NLS$WindowsEnvironmentCodepage is used to set the codepage.

Results:

The result type is INTEGER(4). The result is zero if successful. Otherwise, one of the following error codes (defined in DFNLS.F90) may be returned:

- **NLS$ErrorInvalidLanguage**: *language* is invalid or not supported
- **NLS$ErrorInvalidCountry**: *country* is invalid or is not valid with the language specified
- **NLS$ErrorInvalidCodepage**: *codepage* is invalid or not installed on the system

NLSSetLocale works on installed locales only. Windows NT and Windows 95 support many locales, but these must be installed through the system Windows NT Control Panel/International menu or the Windows 95 Control Panel/Regional Settings menu.

Note that when doing mixed-language programming with Fortran and C, calling **NLSSetLocale** with a codepage other than the default environment Windows codepage causes the codepage in the C run-time library to change by calling C's setmbcp( ) routine with the new codepage. Conversely, changing the C run-time library codepage does not change the codepage in the Fortran NLS library.

Calling **NLSSetLocale** has no effect on the locale used by C programs. The locale set with C's setlocale( ) routine is independent of **NLSSetLocale**.

Calling **NLSSetLocale** with the default environment console codepage, NLS$ConsoleEnvironmentCodepage, causes an implicit call to the Win32 API SetFileApisToOEM( ). Calling **NLSSetLocale** with any other codepage causes a call to SetFileApisToANSI( ).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: NLSGetLocale

NOT

Elemental Intrinsic Function (Generic): Returns the logical complement of the argument.

Syntax

result = **NOT** (*i*)

i
(Input) 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 Model for Bit Data.

Specific Name	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
INOT	INTEGER(2)	INTEGER(2)
JNOT	INTEGER(4)	INTEGER(4)
KNOT [1]	INTEGER(8)	INTEGER(8)
[1] Alpha only		

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IAND, IBCHNG, IBCLR, IBSET, IEOR, IOR, ISHA, ISHC, ISHL, ISHFT, ISHFTC

Examples

If I has a value equal to 10101010 (base 2), NOT (I) has the value 01010101 (base 2).

The following shows another example:

```
INTEGER(2) i(2), j(2)
i = (/4, 132/)          ! i(1) = 0000000000000100
                        ! i(2) = 0000000010000100
j = NOT(i)              ! returns (-5,-133)
                        ! j(1) = 1111111111111011
                        ! j(2) = 1111111101111011
```

NULL

Transformational Intrinsic Function (Generic): Returns a disassociated pointer. This is a Fortran 95 intrinsic function.

Syntax

result = **NULL** ([*mold*])

mold
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.

NULLIFY

Statement: Disassociates a pointer from a target.

Syntax

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**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATE, ASSOCIATED, DEALLOCATE, POINTER, TARGET, NULL, Pointer Assignments.

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.

The following shows another example:

```
!    POINTER2.F90    Pointing at a Pointer and Target
!DEC$ FIXEDFORMLINESIZE:80
```

```
REAL, POINTER :: arrow1 (:)
REAL, POINTER :: arrow2 (:)
REAL, ALLOCATABLE, TARGET :: bullseye (:)

ALLOCATE (bullseye (7))
bullseye = 1.
bullseye (1:7:2) = 10.
WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

arrow1 => bullseye
WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1

arrow2 => arrow1
IF (ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is pointed.'
WRITE (*,'(1x,a,7f8.0)') 'pointer',arrow2

NULLIFY (arrow2)
IF (.NOT.ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is not pointed.'
WRITE (*,'( 1x,a,7f8.0)') 'pointer',arrow1
WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

END
```

NUMBER_OF_PROCESSORS

Inquiry Intrinsic Function (Specific): Returns the total number of processors (peers) available to the program.

Syntax

result = **NUMBER_OF_PROCESSORS** ([*dim*])

dim
(Optional) Has no effect on currently available configurations of DIGITAL systems. This option is provided for compatibility with the High Performance Fortran (HPF) language specification. If *dim* is specified, it must have the value of 1.

Results:

The result type is default integer scalar. The result value is the total number of processors (peers) available to the program.

For single-processor workstations, the result value is 1.

NWORKERS

Inquiry Intrinsic Function (Specific): Returns the number of processes executing a routine. This is a specific function with no generic name. It is provided for compatibility with DIGITAL Fortran 77 for OpenVMS VAX systems.

Syntax

result = **NWORKERS** ( )

Results:

The result is always 1.

OBJCOMMENT

Compiler Directive: Specifies a library search path in an object file.

Syntax

*c***DEC$ OBJCOMMENT** LIB**:***library*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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 in **OBJCOMMENT** 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.

The following form is also allowed: `!MS$OBJCOMMENT LIB:library`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: General Compiler Directives.

Example

```
!  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
```

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.

Syntax

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 FILE Specifier and STATUS 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 (the number of the error message) if an error occurs, a negative integer if an end-of-file record is encountered, and zero if no error occurs. For more information, see I/O Status Specifier.

slist
Is one or more of the following **OPEN** specifiers in the form *specifier = value* or *specifier* (each specifier can appear only once):

ACCESS	CONVERT	MODE	RECORDTYPE
ACTION	DEFAULTFILE	NAME	SHARE
ASSOCIATEVARIABLE	DELIM	ORGANIZATION	SHARED
BLANK	DISPOSE	PAD	STATUS
BLOCKSIZE	FILE	POSITION	TITLE
BUFFERCOUNT	FORM	READONLY	TYPE
BUFFERED	IOFOCUS	RECL	USEROPEN
CARRIAGECONTROL	MAXREC	RECORDSIZE	

The **OPEN** specifiers and their acceptable values are summarized in the OPEN Statement in the *Language Reference*.

The control specifiers that can be specified in an **OPEN** statement are discussed in I/O Control List in the *Language Reference*.

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 **OPEN**s 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, DELIM, 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: READ, WRITE, CLOSE, FORMAT, INQUIRE, OPEN Statement

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*6 FINAL /' '/
```

```
...
IF (expr) FINAL = 'DELETE'
OPEN (UNIT=1, STATUS='NEW', DISP=FINAL)
```

The following statement creates a new sequential formatted file on unit 1 with the default file name fort.1:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The following statement creates a file on magnetic tape:

```
OPEN (UNIT=I, FILE='/dev/rmt8',                                    &
     STATUS='NEW', ERR=14, RECL=1024)
```

The following statement opens the file (created in the previous example) for input:

```
OPEN (UNIT=I, FILE='/dev/rmt8', READONLY, STATUS='OLD',            &
     RECL=1024)
```

The following example opens the existing file /usr/users/someone/test.dat:

```
 OPEN (unit=10, DEFAULTFILE='/usr/users/someone/', FILE='test.dat',
1     FORM='FORMATTED', STATUS='OLD')
```

The following example opens a new file:

```
! Prompt user for a filename and read it:
CHARACTER*64 filename
WRITE (*, '(A\)') ' enter file to create: '
READ (*, '(A)') filename
! Open the file for formatted sequential access as unit 7.
! Note that the specified access need not have been specified,
! since it is the default (as is "formatted").
OPEN (7, FILE = filename, ACCESS = 'SEQUENTIAL', STATUS = 'NEW')
The following example opens an existing file called DATA3.TXT:
! Open a file created by an editor, "DATA3.TXT", as unit 3:
OPEN (3, FILE = 'DATA3.TXT')
```

OPTIONAL

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **OPTIONAL** [, *att-ls*] **::** *d-arg* [, *d-arg*]...

Statement:

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.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRESENT, Argument Keywords in Intrinsic Procedures, Optional Arguments, Argument Association, Type Declarations, Compatible attributes

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.

The following shows another example:

```
SUBROUTINE ADD (a,b,c,d)
REAL              a, b, d
REAL, OPTIONAL :: c

IF (PRESENT(c)) THEN
  d = a + b + c + d
ELSE
  d = a + b + d
END IF
END SUBROUTINE
```

Consider the following:

```
SUBROUTINE EX (a, b, c)
REAL, OPTIONAL :: b,c
```

This subroutine can be called with any of the following statements:

```
CALL EX (x, y, z)   !All 3 arguments are passed.

CALL EX (x)         !Only the first argument is passed.

CALL EX (x, c=z)    !The first optional argument is omitted.
```

Note that you *cannot* use a series of commas to indicate omitted optional arguments, as in the

following example:

```
CALL EX (x,,z)   !Invalid statement.
```

This results in a compile-time error.

OPTIONS Directive

Compiler Directive: 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.

Syntax

*c***DEC$ OPTIONS /**[**NO**]**ALIGN**[= *p*]
...
*c***DEC$ END OPTIONS**

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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 90 standard).
  This keyword causes the compiler to naturally align all data in a common block, including INTEGER(KIND=8), REAL(KIND=8), and all COMPLEX data.
- **STANDARD**
  Naturally aligns data items in common blocks on up to 32-bit boundaries (consistent with the Fortran 90 standard).

This keyword only applies to common blocks; so, you can specify **OPTIONS** /ALIGN=COMMONS=STANDARD, but you cannot specify /ALIGN=STANDARD.

ALL
Is the same as specifying **OPTIONS** /ALIGN, **OPTIONS** /ALIGN=NATURAL, and **OPTIONS** /ALIGN=(RECORDS=NATURAL,COMMONS=NATURAL).

NONE
Is the same as specifying **OPTIONS** /NOALIGN, **OPTIONS** /ALIGN=PACKED, and **OPTIONS** /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 /alignment compiler option.

For performance reasons, DIGITAL Fortran aligns local data items on natural boundaries. However, **EQUIVALENCE**, **COMMON**, **RECORD**, and **STRUCTURE** data declaration statements can force misaligned data. By default, you receive compiler messages when 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 *c***DEC$ OPTIONS/ALIGN** (or the /alignment compiler option) minimizes misaligned data.

If you want aligned data in common blocks, do one of the following:

- Specify **OPTIONS** /ALIGN=COMMONS=STANDARD for data items up to 32 bits in length.
- Specify **OPTIONS** /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 **OPTIONS** /ALIGN=RECORDS=PACKED.
- Place source data declarations in the record structure so that the data is naturally aligned.

See Also: General Compiler Directives

Example

```
! directives can be nested up to 100 levels
  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).

OPTIONS

Statement: Overrides or confirms the compiler options in effect for a program unit.

Syntax

OPTIONS *option* [option...]

option
Is one of the following:

/ASSUME =	[NO]UNDERSCORE (Alpha only)
/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 (VMS only) G_FLOAT (VMS only) IEEE_FLOAT
/[NO]G_FLOATING (VMS only)	
/[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.

Example

The following are valid **OPTIONS** statements:

```
OPTIONS /CHECK=ALL/F77
OPTIONS /I4
```

OR

Elemental Intrinsic Function: Performs a bitwise inclusive **OR** on its arguments. For more information, see IOR.

Example

```
INTEGER i
i = OR(3, 10)   ! returns 11
```

OUTGTEXT

Graphics Subroutine: In graphics mode, sends a string of text to the screen, including any trailing blanks.

Module: USE DFLIB

Syntax

CALL OUTGTEXT (*text*)

text
(Input) Character*(*). String to be displayed.

Text output begins at the current graphics position, using the current font set with **SETFONT** and the current color set with **SETCOLORRGB** or **SETCOLOR**. No formatting is provided. After it outputs the text, **OUTGTEXT** updates the current graphics position.

Before you call **OUTGTEXT**, you must call **INITIALIZEFONTS**.

Because **OUTGTEXT** is a graphics function, the color of text is affected by the **SETCOLORRGB** function, not by **SETTEXTCOLORRGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETFONTINFO, GETGTEXTEXTENT, INITIALIZEFONTS, MOVETO, SETCOLORRGB, SETFONT, SETGTEXTROTATION

Example

```
! build as a QuickWin App.
USE DFLIB
INTEGER(2) result
INTEGER(4) i
TYPE (xycoord) xys

result = INITIALIZEFONTS()
result = SETFONT('t''Arial''h18w10pvib')
do i=1,6
   CALL MOVETO(INT2(0),INT2(30*(i-1)),xys)
   grstat=SETCOLOR(INT2(i))
   CALL OUTGTEXT('This should be ')
   SELECT CASE (i)
     CASE (1)
       CALL OUTGTEXT('Blue')
     CASE (2)
       CALL OUTGTEXT('Green')
     CASE (3)
       CALL OUTGTEXT('Cyan')
     CASE (4)
       CALL OUTGTEXT('Red')
     CASE (5)
       CALL OUTGTEXT('Magenta')
     CASE (6)
       CALL OUTGTEXT('Orange')
   END SELECT
END DO
END
```

OUTTEXT

Graphics Subroutine: In text or graphics mode, sends a string of text to the screen, including any trailing blanks.

Module: USE DFLIB

Syntax

CALL OUTTEXT (*text*)

text
(Input) Character*(*). String to be displayed.

Text output begins at the current text position in the color set with **SETTEXTCOLORRGB** or **SETTEXTCOLOR**. No formatting is provided. After it outputs the text, **OUTTEXT** updates the current text position.

To output text using special fonts, you must use the **OUTGTEXT** function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETTEXTPOSITION, SETTEXTCOLORRGB, WRITE

Example

```
USE DFLIB
INTEGER(2) oldcolor
TYPE (rccoord) rc

CALL CLEARSCREEN($GCLEARSCREEN)
CALL SETTEXTPOSITION (INT2(1), INT2(5), rc)
oldcolor = SETTEXTCOLOR(INT2(4))
CALL OUTTEXT ('Hello, everyone')
END
```

PACK Directive

Compiler Directive: Specifies the memory starting addresses of derived-type items (and record structure items).

Syntax

*c***DEC$ PACK**:[{1 | 2 | 4}]

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

Rules and Behavior

Items of derived types, unions, and 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 /alignment or /vms 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 C**DEC$ 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 C**DEC$ 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.

The following form is also allowed: `!MS$PACK:[{1|2|4}]`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Derived Type, STRUCTURE...END STRUCTURE, UNION...END UNION, General Compiler Directives.

Example

```
! 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:
```

PACK

Transformational Intrinsic Function (Generic): Takes elements from an array and packs them into a rank-one array under the control of a mask.

Syntax

result = **PACK** (*array*, *mask* [, *vector*])

array
(Input) Must be an array (of any data type).

mask
(Input) Must be of type logical and conformable with *array*. It determines which elements are

taken from *array*.

vector
(Optional; input) 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: UNPACK

Examples

N is the array

```
[ 0  8  0 ]
[ 0  0  0 ]
[ 7  0  0 ].
```

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).

The following shows another example:

```
INTEGER array(2, 3), vec1(2), vec2(5)
LOGICAL mask (2, 3)
array = RESHAPE((/7, 0, 0, -5, 0, 0/), (/2, 3/))
mask = array .NE. 0
!  array is  7  0 0  and mask is  T F F
!            0 -5 0               F T F

VEC1 = PACK(array, mask)      ! returns ( 7, -5 )
VEC2 = PACK(array, array .GT. 0, VECTOR= (/1,2,3,4,5/))
!  returns ( 7, 2, 3, 4, 5 )
```

PACKTIMEQQ

Run-Time Subroutine: Packs time and date values.

Module: USE DFLIB

Syntax

CALL PACKTIMEQQ (*timedate*, *iyr*, *imon*, *iday*, *ihr*, *imin*, *isec*)

timedate
(Output) INTEGER(4). Packed time and date information.

iyr
(Input) INTEGER(2). Year (*xxxx* AD).

imon
(Input) INTEGER(2). Month (1 - 12).

iday
(Input) INTEGER(2). Day (1 - 31)

ihr
(Input) INTEGER(2). Hour (0 - 23)

imin
(Input) INTEGER(2). Minute (0 - 59)

isec
(Input) INTEGER(2). Second (0 - 59)

The packed time is the number of seconds since 00:00:00 Greenwich mean time, January 1, 1970. Because packed time values can be numerically compared, you can use **PACKTIMEQQ** to work with relative date and time values. Use **UNPACKTIMEQQ** to unpack time information. **SETFILETIMEQQ** uses packed time.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: UNPACKTIMEQQ, SETFILETIMEQQ, GETFILEINFOQQ, TIME

Example

```
USE DFLIB
INTEGER(2) year, month, day, hour, minute, second,   &
hund
INTEGER(4) timedate
```

```
CALL GETDAT (year, month, day)
CALL GETTIM (hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day, hour,   &
                 minute, second)
END
```

PARAMETER

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **PARAMETER** [, *att-ls*] **::** *c* = *expr* [, *c* = *expr*] ...

Statement:

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.

The use of parentheses is optional and can be controlled using the /[no]altparam compiler option.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATA, Type Declarations, Using the Compiler and Linker from the Command Line, Compatible attributes, Initialization Expressions, IMPLICIT, Alternative syntax for the PARAMETER statement.

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')
```

The following shows another example:

```
! implicit integer type
PARAMETER (nblocks = 10)

! implicit real type
IMPLICIT REAL (L-M)
PARAMETER (loads = 10.0, mass = 32.2)

! typed by PARAMETER statement
! Requires compiler option
PARAMETER mass = 47.3, pi = 3.14159
PARAMETER bigone = 'This constant is larger than forty characters'

! PARAMETER in attribute syntax
REAL, PARAMETER :: mass=47.3, pi=3.14159, loads=10.0, mass=32.2
```

PASSDIRKEYSQQ

QuickWin Function: Determines the behavior of direction and page keys in a QuickWin application.

Module: USE DFLIB

Syntax

result = **PASSDIRKEYSQQ** (*val*)

val
A value of .TRUE. causes direction and page keys to be input as normal characters (the PassDirKeys flag is turned on). A value of .FALSE. causes direction and page keys to be used for scrolling.

Results:

The return value indicates the previous setting of the PassDirKeys flag.

When the PassDirKeys flag is turned on, the mouse must be used for scrolling since the direction and page keys are treated as normal input characters.

The **PASSDIRKEYSQQ** function is meant to be used primarily with the **GETCHARQQ** and **INCHARQQ** functions. Do *not* use normal input statements (such as **READ**) with the PassDirKeys flag turned on, unless your program is prepared to interpret direction and page keys.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCHARQQ, INCHARQQ.

Example

```
use dflib

logical*4 res
character*1 ch, ch1

Print *,"Type X to exit, S to scroll, D to pass Direction keys"

123 continue
ch = getcharqq( )
! check for escapes
! 0x00 0x?? is a function key
! 0xE0 0x?? is a direction key
if (ichar(ch) .eq. 0) then
  ch1 = getcharqq()
  print *,"function key follows escape = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
else if (ichar(ch) .eq. 224) then
  ch1 = getcharqq()
  print *,"direction key follows escape = ",ichar(ch),"  ",ichar(ch1),"  ",ch1
  goto 123
else
  print *,ichar(ch),"  ",ch

  if(ch .eq. 'S') then
   res = passdirkeysqq(.false.)
   print *, "Entering Scroll mode ",res
  endif

  if(ch .eq. 'D')  then
   res = passdirkeysqq(.true.)
   print *, "Entering Direction keys mode ",res
  endif

  if(ch .ne. 'X')  go to 123

endif
end
```

PAUSE

Statement: Temporarily suspends program execution and lets you execute operating system commands during the suspension. The **PAUSE** statement is an obsolescent feature in Fortran 90, which has been deleted in Fortran 95. DIGITAL Fortran fully supports features deleted in Fortran 95.

Syntax

PAUSE [*pause-code*]

pause-code
(Optional) 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 following prompt is then displayed:

- On Windows NT and Windows 95 systems:

  ```
  Fortran Pause - Enter command<CR> or <CR> to continue.
  ```

- On OpenVMS systems, the system prompt
- On DIGITAL UNIX systems:

  ```
  PAUSE prompt>
  ```

Effect on Windows NT and Windows 95 Systems

The program waits for input on `stdin`. If you enter a blank line, execution resumes at the next executable statement.

Anything else is treated as a DOS command and is executed by a system( ) call. The program loops, letting you execute multiple DOS commands, until a blank line is entered. Execution then resumes at the next executable statement.

Effect on OpenVMS Systems

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.

Effect on DIGITAL UNIX Systems

The effect of PAUSE differs depending on whether the program is a foreground or background process, as follows:

- If a program is a foreground process, the program is suspended until you enter the **CONTINUE** command. Execution then resumes at the next executable statement.

  Any other command terminates execution.

- If a program is a background process, the behavior depends on `stdin`, as follows:

  - If `stdin` is redirected from a file, the system displays the following (after the pause code and prompt):

    ```
    To continue from background, execute 'kill -15 n'
    ```

    In this message, `n` is the process id of the program.

  - If `stdin` is not redirected from a file, the program becomes a suspended background job, and you must specify `fg` to bring the job into the foreground. You can then enter a command to resume or terminate processing.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: STOP, SYSTEM, Obsolescent and Deleted Language Features

Examples

The following examples show valid **PAUSE** statements:

```
  PAUSE 701
  PAUSE 'ERRONEOUS RESULT DETECTED'
```

The following shows another example:

```
  CHARACTER*24 filename
  PAUSE 'Enter DIR to see available files or press RETURN'   &
 &' if you already know filename.'
  READ(*,'(A\)') filename
  OPEN(1, FILE=filename)
  . . .
```

PEEKCHARQQ

Run-Time Function: Checks the keystroke buffer for a recent console keystroke and returns .TRUE. if there is a character in the buffer or .FALSE. if there is not.

Module: USE DFLIB

Syntax

result = **PEEKCHARQQ** ( )

Results:

The result type is LOGICAL(4). The result is .TRUE. if there is a character waiting in the keyboard buffer; otherwise, .FALSE..

To find out the value of the key in the buffer, call **GETCHARQQ**. If there is no character waiting in the buffer when you call **GETCHARQQ**, **GETCHARQQ** waits until there is a character in the buffer. If you call **PEEKCHARQQ** first, you prevent **GETCHARQQ** from halting your process while it waits for a keystroke. If there is a keystroke, **GETCHARQQ** returns it and resets **PEEKCHARQQ** to .FALSE..

Compatibility

CONSOLE DLL LIB

See Also: GETCHARQQ, INCHARQQ, GETSTRQQ, FGETC, GETC

Example

```
USE DFLIB
LOGICAL(4) pressed / .FALSE. /

DO WHILE (.NOT. pressed)
  WRITE(*,*) ’ Press any key’
  pressed = PEEKCHARQQ ( )
END DO
END
```

PERROR

Portability Subroutine: Sends a message to the standard error stream, preceded by a specified string, for the last detected error.

Module: USE DFPORT

Syntax

CALL PERROR (*string*)

string

(Input) Character*(*). Message to precede the standard error message.

The string sent is the same as that given by **GERROR**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also:R GERROR, IERRNO

Example

```
USE DFPORT
character*24 errtext
errtext = 'In my opinion, '
. . .
! any error message generated by errtext is
! preceded by 'In my opinion, '
Call PERROR (errtext)
```

PIE, PIE_W

Graphics Function: Draws a pie-shaped wedge in the current graphics color.

Module: USE DFLIB

Syntax

result = **PIE** (*i*, *x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)
result = **PIE_W** (*i*, *wx1*, *wy1*, *wx2*, *wy2*, *wx3*, *wy3*, *wx4*, *wy4*)

i
(Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GFILLINTERIOR**: Fills the figure using the current color and fill mask.
- **$GBORDER**: Does not fill the figure.

x1, *y1*
(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

x2, *y2*
(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

x3, *y3*
(Input) INTEGER(2). Viewport coordinates of start vector.

x4, *y4*
(Input) INTEGER(2). Viewport coordinates of end vector.

wx1, *wy1*
(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

wx3, *wy3*
(Input) REAL(8). Window coordinates of start vector.

wx4, *wy4*
(Input) REAL(8). Window coordinates of end vector.

Results:

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the pie is clipped or partially out of bounds, the pie is considered successfully drawn and the return is 1. If the pie is drawn completely out of bounds, the return is 0.

The border of the pie wedge is drawn in the current color set by **SETCOLORRGB**.

The **PIE** function uses the viewport-coordinate system. The center of the arc is the center of the bounding rectangle, which is specified by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*).

The **PIE_W** function uses the window-coordinate system. The center of the arc is the center of the bounding rectangle specified by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*wx3*, *wy3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*wx4*, *wy4*).

The fill flag option $GFILLINTERIOR is equivalent to a subsequent call to **FLOODFILLRGB** using the center of the pie as the starting point and the current graphics color (set by **SETCOLORRGB**) as the fill color. If you want a fill color different from the boundary color, you cannot use the $GFILLINTERIOR option. Instead, after you have drawn the pie wedge, change the current color with **SETCOLORRGB** and then call **FLOODFILLRGB**. You must supply **FLOODFILLRGB** with an interior point in the figure you want to fill. You can get this point for the last drawn pie or arc by calling **GETARCINFO**.

If you fill the pie with **FLOODFILLRGB**, the pie must be bordered by a solid line style. Line style is solid by default and can be changed with **SETLINESTYLE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETCOLORRGB, SETFILLMASK, SETLINESTYLE, FLOODFILLRGB,

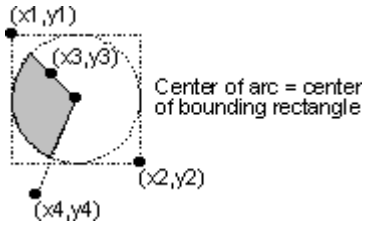
GETARCINFO, ARC, ELLIPSE, GRSTATUS, LINETO, POLYGON, RECTANGLE

Example

```
! build as Graphics App.
USE DFLIB
INTEGER(2) status, dummy
INTEGER(2) x1, y1, x2, y2, x3, y3, x4, y4
x1 = 80; y1 = 50
x2 = 180; y2 = 150
x3 = 110; y3 = 80
x4 = 90; y4 = 180

status = SETCOLOR(INT2(4))
dummy = PIE( $GFILLINTERIOR, x1, y1, x2, y2, &
                             x3, y3, x4, y4)
END
```

Figure: Coordinates Used to Define PIE and PIE_W



POINTER -- Fortran 90

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **POINTER** [, *att-ls*] **::** *ptr* [(*d-spec*)] [, *ptr* [(*d-spec*)]]...

Statement:

POINTER [**::**] *ptr* [(*d-spec*)] [, *ptr* [(*d-spec*)]]...

type-spec
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
(Optional) 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.

Fortran 90 pointers are not the same as integer pointers. For more information, see the POINTER -- DIGITAL Fortran statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATE, ASSOCIATED, DEALLOCATE, NULLIFY, TARGET, Deferred-Shape Arrays, Pointer Assignments, Pointer Association, Pointer Arguments, NULL, DIGITAL Fortran POINTER statement, Type Declarations, Compatible attributes.

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(:,:)
```

See also the examples `POINTER.F90` and `POINTER2.F90` in /DF98/SAMPLES/TUTORIAL.

The following shows another example:

```
REAL, POINTER :: arrow (:)
REAL, ALLOCATABLE, TARGET :: bullseye (:,:)

! The following statement associates the pointer with an unused
! block of memory.

ALLOCATE (arrow (1:8), STAT = ierr)
IF (ierr.eq.0) WRITE (*,'(/1x,a)') 'ARROW allocated'
arrow = 5.
WRITE (*,'(1x,8f8.0/)') arrow
ALLOCATE (bullseye (1:8,3), STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'BULLSEYE allocated'
bullseye = 1.
bullseye (1:8:2,2) = 10.
WRITE (*,'(1x,8f8.0)') bullseye

! The following association breaks the association with the first
! target, which being unnamed and unassociated with other pointers,
! becomes lost. ARROW acquires a new shape.

arrow => bullseye (2:7,2)
WRITE (*,'(/1x,a)') 'ARROW is repointed & resized, all the 5s are lost'
WRITE (*,'(1x,8f8.0)') arrow

NULLIFY (arrow)
IF (.NOT.ASSOCIATED(arrow)) WRITE (*,'(/a/)') ' ARROW is not pointed'

DEALLOCATE (bullseye, STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'Deallocation successful.'
END
```

POINTER -- DIGITAL Fortran

Statement: Establishes pairs of variables and pointers, in which each pointer contains the address of its paired variable. This statement is different from the Fortran 90 POINTER statement.

Syntax

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. On Windows NT and Windows 95 systems, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER (4)).
- 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

  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
```

- 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 (DIGITAL Fortran) **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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LOC, MALLOC, FREE

Example

```
POINTER (p, k(2))
INTEGER j(2)

! This has the same effect as j(1) = 0, j(2) = 5
p = LOC(j)
k(1) = 0
p = p + 4 ! for 4-byte integer
k(2) = 5
```

POLYGON, POLYGON_W

Graphics Function: Draws a polygon using the current graphics color, logical write mode, and line style.

Module: USE DFLIB

Syntax

result = **POLYGON** (*control*, *ppoints*, *cpoints*)
result = **POLYGON_W** (*control*, *wppoints*, *cpoints*)

control
(Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GFILLINTERIOR**: Draws a solid polygon using the current color and fill mask.
- **$GBORDER**: Draws the border of a polygon using the current color and line style.

ppoints
(Input) Derived type xycoord. Array of derived types defining the polygon vertices in viewport coordinates. The derived type xycoord is defined in DFLIB.F90 as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints
(Input) INTEGER(2). Number of polygon vertices.

wppoints (Input) Derived type wxycoord. Array of derived types defining the polygon vertices in window coordinates. The derived type wxycoord is defined in DFLIB.F90 as follows:

```
TYPE wxycoord
  REAL(8) wx
  REAL(8) wy
END TYPE wxycoord
```

Results:

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

The border of the polygon is drawn in the current graphics color, logical write mode, and line style, set with **SETCOLORRGB**, **SETWRITEMODE**, and **SETLINESTYLE**, respectively. The **POLYGON** routine uses the viewport-coordinate system (expressed in xycoord derived types), and the **POLYGON_W** routine uses real-valued window coordinates (expressed in wxycoord types).

The arguments *ppoints* and *wppoints* are arrays whose elements are xycoord or wxycoord derived types. Each element specifies one of the polygon's vertices. The argument *cpoints* is the number of elements (the number of vertices) in the *ppoints* or *wppoints* array.

Note that **POLYGON** draws between the vertices in their order in the array. Therefore, when drawing outlines, skeletal figures, or any other figure that is not filled, you need to be careful about the order of the vertices. If you don't want lines between some vertices, you may need to repeat vertices to make the drawing backtrack and go to another vertex to avoid drawing across your figure. Also, **POLYGON** draws a line from the last specified vertex back to the first vertex.

If you fill the polygon using **FLOODFILLRGB**, the polygon must be bordered by a solid line style. Line style is solid by default and can be changed with **SETLINESTYLE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETCOLORRGB, SETFILLMASK, SETLINESTYLE, FLOODFILLRGB, GRSTATUS, LINETO, RECTANGLE, SETWRITEMODE

Example

```
! Build as a Graphics App.
 !
 ! Draw a skeletal box
     USE DFLIB

     INTEGER(2) status
     TYPE (xycoord) poly(12)

 ! Set up box vertices in order they will be drawn, &
 ! repeating some to avoid unwanted lines across box

     poly(1).xcoord = 50
     poly(1).ycoord = 80
     poly(2).xcoord = 85
     poly(2).ycoord = 35
     poly(3).xcoord = 185
     poly(3).ycoord = 35
     poly(4).xcoord = 150
     poly(4).ycoord = 80
     poly(5).xcoord = 50
     poly(5).ycoord = 80
     poly(6).xcoord = 50
     poly(6).ycoord = 180
     poly(7).xcoord = 150
     poly(7).ycoord = 180
     poly(8).xcoord = 185
     poly(8).ycoord = 135
     poly(9).xcoord = 185
     poly(9).ycoord = 35
     poly(10).xcoord = 150
     poly(10).ycoord = 80
     poly(11).xcoord = 150
     poly(11).ycoord = 180
     poly(12).xcoord = 150
     poly(12).ycoord = 80

     status = SETCOLORRGB(#0000FF)
     status = POLYGON($GBORDER, poly, INT2(12))
     END
```

POPCNT

Elemental Intrinsic Function (Generic): Returns the number of 1 bits in an integer.

Syntax

result = **POPCNT** (*i*)

i
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 Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPCNT(I) is 5.

POPPAR

Elemental Intrinsic Function (Generic): Returns the parity of an integer.

Syntax

result = **POPPAR** (*i*)

i
Integer.

Results:

The result type is the same as *i*. The result value is one if there are an odd number of 1 bits in the binary representation of the integer *i* and zero if there are an even number.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPPAR(I) is 1.

PRECISION

Inquiry Intrinsic Function (Generic): Returns the decimal precision in the model representing real numbers with the same kind parameter as the argument.

Syntax

result = **PRECISION** (*x*)

x
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

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 Model for Real Data.

PRESENT

Inquiry Intrinsic Function (Generic): Returns whether or not an optional dummy argument is present (has an associated actual argument).

Syntax

result = **PRESENT** (*a*)

a
(Input) Must be an optional argument of the current procedure.

Results:

The result type is default logical scalar. The result is .TRUE. if *a* is present; otherwise, the result is .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: OPTIONAL

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
```

The following shows another example:

```
CALL who( 1, 2 ) ! prints "A present" "B present"
CALL who( 1 ) ! prints "A present"
CALL who( b = 2 ) ! prints "B present"
CALL who( ) ! prints nothing
CONTAINS
  SUBROUTINE who( a, b )
    INTEGER(4), OPTIONAL :: a, b
    IF (PRESENT(a)) PRINT *,'A present'
    IF (PRESENT(b)) PRINT *,'B present'
  END SUBROUTINE who
END
```

PRINT

Statement: Displays output on the screen. **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:

Syntax

Formatted

PRINT *form* [, *io-list*]

Formatted: List-Directed

PRINT * [, *io-list*]

Formatted: Namelist

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PUTC, READ, WRITE, FORMAT, Data Transfer I/O Statements, File Operation I/O Statements

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)
```

The following shows another example:

```
!   The following statements are equivalent:
    PRINT     '(A11)', 'Abbottsford'
    WRITE (*, '(A11)') 'Abbottsford'
    TYPE      '(A11)', 'Abbottsford'
```

PRIVATE

Statement and Attribute: Specifies that entities in a module can be accessed only within the module itself.

The PRIVATE attribute can be specified in a type declaration statement or a **PRIVATE** statement, and takes one of the following forms:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **PRIVATE** [, *att-ls*] **::** *entity* [, *entity*]...

Statement:

PRIVATE [[**::**] *entity* [, *entity*]...]

type
Is a data type specifier.

att-ls
Is an optional list of attribute specifiers.

entity

Is one of the following:

- A variable name
- A procedure name
- A derived type name
- A named constant
- A 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 attribute can only appear in the scoping unit of a module.

Only one **PRIVATE** 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 **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 they are not accessible from outside the module.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: MODULE, PUBLIC, TYPE, Defining Generic Names for Procedures, USE, Use and Host Association, Type Declarations, Compatible attributes.

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.

The following shows another example:

```
 !LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 !The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
        REAL,INTENT(OUT) z
        CALL SQUARE(x,y)
        z = SQRT(x + y)
        RETURN
     END SUBROUTINE
     SUBROUTINE SQUARE(x1,y1)
        REAL x1,y1
        x1 = x1**2
        y1 = y1**2
```

```
    RETURN
  END SUBROUTINE
END MODULE
```

PROCESSORS_SHAPE

Inquiry Intrinsic Function (Specific): Returns the shape of an implementation-dependent hardware processor array.

Syntax

result = **PROCESSORS_SHAPE** ( )

Results:

The result is a rank-one array of size zero.

This function is provided for compatibility with High Performance Fortran.

PRODUCT

Transformational Intrinsic Function (Generic): Returns the product of all the elements in an entire array or in a specified dimension of an array.

Syntax

result = **PRODUCT** (*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer or real.

dim
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.

mask
(Optional; input) 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 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 there are no true elements, 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)]).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SUM

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

```
[ 1  4  7 ]
[ 2  3  5 ].
```

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.

If *array* has shape (2, 2, 2), *mask* is omitted, and *dim* is 1, the result is an array result with shape (2, 2) whose elements have the following values.

Resultant array element	Value
result(1, 1)	*array*(1, 1, 1) * *array*(2, 1, 1)
result(2, 1)	*array*(1, 2, 1) * *array*(2, 2, 1)
result(1, 2)	*array*(1, 1, 2) * *array*(2, 1, 2)
result(2, 2)	*array*(1, 2, 2) * *array*(2, 2, 2)

The following shows another example:

```
INTEGER array (2, 3)
INTEGER AR1(3), AR2(2)
array = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2,3/))
! array is  1 2 3
!           4 5 6

AR1 = PRODUCT(array, DIM = 1) ! returns [ 4 10 18 ]
AR2 = PRODUCT(array, MASK = array .LT. 6, DIM = 2)
                                  ! returns [ 6 20 ]
END
```

PROGRAM

Statement: Identifies the program unit as a main program and gives it a name.

Syntax

[**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.

If a name follows the **END** statement, it must be the same as the name specified in the **PROGRAM** statement. The program name 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Building Programs and Libraries

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
```

The following shows another example:

```
 PROGRAM MyProg
 PRINT *, 'hello world'
 END
```

PSECT

Compiler Directive: Modifies several characteristics of a common block.

Syntax

*c***DEC$ PSECT** /*common-name*/ *a* [, *a*]...

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

common-name
Is the name of the common block. The slashes (/) are required.

a
Is one of the following:

ALIGN=*val* or ALIGN=*keyword*

Specifies alignment for the common block.

The *val* must be a constant ranging from 0 through 16 on VMS systems, 0 through 6 on Windows NT and Windows 95 systems, and 0 through 4 on DIGITAL UNIX systems. 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
PAGE (VMS only)[1]	16

[1] On DIGITAL UNIX, Windows NT and Windows 95 systems, this keyword produces an error.

Rules and Behavior

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. The following table shows the default common block alignment and how it can be modified by **PSECT**:

Default Alignment	PSECT Modification
On Intel processors:	
Octaword alignment [1] (4)	0 through 6 [2]
On Alpha processors:	
Octaword alignment [1] (4)	VMS: 0 through 16 [3] Windows NT: 0 through 6 [2] DIGITAL UNIX: 0 through 4 [2]

[1] An address that is an integral multiple of 16.
[2] Or keywords BYTE through OCTA.
[3] Or keywords BYTE through PAGE.

See Also: General Compiler Directives.

PUBLIC

Statement and Attribute: Specifies that entities in a module can be accessed from outside the module (by specifying a **USE** statement).

The PUBLIC attribute can be specified in a type declaration statement or a **PUBLIC** statement, and takes one of the following forms:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **PUBLIC** [, *att-ls*] **::** *entity* [, *entity*]...

Statement:

PUBLIC [[**::**] *entity* [, *entity*]...]

type
Is a data type specifier.

att-ls
Is an optional list of attribute specifiers.

entity
Is one of the following:

- A variable name

- A procedure name
- A derived type name
- A named constant
- A 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 PUBLIC attribute can only appear in the scoping unit of a module.

Only one **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 PRIVATE statements are specified in a module, the default is PUBLIC accessibility.

If a 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRIVATE, MODULE, TYPE, Defining Generic Names for Procedures, USE, Use and Host Association, Type Declarations, Compatible attributes.

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 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.

The following shows another example:

```
 !LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 !The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
        REAL,INTENT(OUT) z
        CALL SQUARE(x,y)
        z = SQRT(x + y)
        RETURN
     END SUBROUTINE
     SUBROUTINE SQUARE(x1,y1)
        REAL x1,y1
        x1 = x1**2
        y1 = y1**2
        RETURN
     END SUBROUTINE
    END MODULE
```

PURE

Keyword: Asserts that a user-defined procedure has no side effects. This kind of procedure is specified by using the prefix **PURE** (or **ELEMENTAL**) in a **FUNCTION** or **SUBROUTINE** statement. Pure procedures are a Fortran 95 feature.

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**

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.

See Also: FUNCTION, SUBROUTINE, FORALL, ELEMENTAL prefix

Examples

Consider the following:

```
PURE FUNCTION DOUBLE(X)
  REAL, INTENT(IN) :: X
  DOUBLE = 2 * X
END FUNCTION DOUBLE
```

The following shows another example:

```
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
```

PUTC

Portability Function: Writes a character to Fortran external unit number 6.

Module: USE DFPORT

Syntax

result = **PUTC** (*char*)

char
(Input) Character. Character to be written to external unit 6.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETC, WRITE, PRINT, FPUTC, Portability Library

Example

```
use dfport
integer(4) i4
character*1 char1
do i = 1,26
  char1 = char(123-i)
  i4 = putc(char1)
  if (i4.ne.0) iflag = 1
enddo
```

PUTIMAGE, PUTIMAGE_W

Graphics Subroutine: Transfers the image stored in memory to the screen.

Module USE DFLIB

Syntax

CALL PUTIMAGE (*x*, *y*, *image*, *action*)
CALL PUTIMAGE_W (*wx*, *wy*, *image*, *action*)

x, y
(Input) INTEGER(2). Viewport coordinates for upper-left corner of the image when placed on the screen.

wx, wy
(Input) REAL(8). Window coordinates for upper-left corner of the image when placed on the screen.

image
(Input) INTEGER(1). Array of single-byte integers. Stored image buffer.

action
(Input) INTEGER(2). Interaction of the stored image with the existing screen image. One of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GAND**: Forms a new screen display as the logical AND of the stored image and the existing screen display. Points that have the same color in both the existing screen image and the stored image remain the same color, while points that have different colors are joined by a logical AND.
- **$GOR**: Superimposes the stored image onto the existing screen display. The resulting image is the logical OR of the image.
- **$GPRESET**: Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by **GETIMAGE**, producing a negative image.
- **$GPSET**: Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by **GETIMAGE**.
- **$GXOR**: Causes points in the existing screen image to be inverted wherever a point exists in the stored image. This behavior is like that of a cursor. If you perform an exclusive OR of an image with the background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The **$GXOR** constant is a special mode often used for animation.
- In addition, the following ternary raster operation constants can be used (described in the online documentation for the WIN32 API BitBlt):
  - **$GSRCCOPY** (same as **$GPSET**)
  - **$GSRCPAINT** (same as **$GOR**)
  - **$GSRCAND** (same as **$GAND**)
  - **$GSRCINVERT** (same as **$GXOR**)
  - **$GSRCERASE**
  - **$GNOTSRCCOPY** (same as **$GPRESET**)
  - **$GNOTSRCERASE**
  - **$GMERGECOPY**
  - **$GMERGEPAINT**
  - **$GPATCOPY**
  - **$GPATPAINT**
  - **$GPATINVERT**

- **$GDSTINVERT**
- **$GBLACKNESS**
- **$GWHITENESS**

PUTIMAGE places the upper-left corner of the image at the viewport coordinates (*x*,*y*).
PUTIMAGE_W places the upper-left corner of the image at the window coordinates (*wx*, *wy*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETIMAGE, GRSTATUS, IMAGESIZE

Example

```
! Build as a Graphics App.
USE DFLIB
INTEGER(1), ALLOCATABLE :: buffer(:)
INTEGER(2) status, x
INTEGER(4) imsize

status = SETCOLOR(INT2(4))
! draw a circle
status = ELLIPSE($GFILLINTERIOR,INT2(40),INT2(55),    &
                 INT2(70),INT2(85))
imsize = IMAGESIZE (INT2(39),INT2(54),INT2(71), &
                 INT2(86))
ALLOCATE (buffer(imsize))
CALL GETIMAGE(INT2(39),INT2(54),INT2(71),INT2(86),    &
              buffer)
! copy a row of circles beneath it
DO x = 5 , 395, 35
   CALL PUTIMAGE(x, INT2(90), buffer, $GPSET)
END DO
DEALLOCATE(buffer)
END
```

QEXT (VMS and U*X)

Elemental Intrinsic Function (Generic): Converts a number to quad precision (REAL(16)) type.

Syntax

result = **QEXT** (*a*)

a
(Input) Must be of type integer, real, or complex.

Results:

The result type is 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)
	INTEGER(4)	REAL(16)
	INTEGER(8)	REAL(16)
QEXT	REAL(4)	REAL(16)
QEXTD	REAL(8)	REAL(16)
	REAL(16)	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).

QFLOAT (VMS and U*X)

Elemental Intrinsic Function (Generic): Converts an integer to quad precision (REAL(16)) type.

Syntax

result = **QFLOAT** (*a*)

a
(Input) Must be of type integer.

Results:

The result type is 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).

QSORT

Portability Subroutine: Performs a quick sort on an array of rank one.

Module: USE DFPORT

Syntax

CALL QSORT (*array*, *len*, *isize*, *compar*)

array
(Input) Any type. One-dimensional array to be sorted.

len
(Input) INTEGER(4). Number of elements in *array*.

isize
(Input) INTEGER(4). Size, in bytes, of a single element of *array*:

- 4 if *array* is of type REAL(4)
- 8 if *array* is REAL(8) or complex
- 16 if *array* is COMPLEX(8)

compar
(Input) INTEGER(2). Name of a user-defined ordering function that determines sort order. The type declaraton of *compar* takes the form:

INTEGER(2) FUNCTION *compar*(*arg1*, *arg2*)

where *arg1* and *arg2* have the same type as *array*. Once you have created an ordering scheme, implement your sorting function so that it returns the following:

- Negative if *arg1* should precede *arg2*
- Zero if *arg1* is equivalent to *arg2*
- Positive if *arg1* should follow *arg2*

Dummy argument *compar* must be declared as external.

If you use **QSORT** several times with different data types, your program must have a **USE DFPORT** statement in order for all the calls to work correctly. In addition, if you wish to use **QSORT** with a derived type, you must include an overload for the generic subroutine **QSORT**. Examples of how to do this are in the portability module's source file, DFPORT.F90, located in your \DF98\INCLUDE subdirectory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
PROGRAM SORTQ
   USE DFPORT
   integer(2), external :: cmp_function
   integer(2) insort(26), i
   integer (4) array_len, array_size
   array_len = 26
   array_size = 2
   do i=90,65,-1
     insort(i-64)=91 - i
   end do
   print *, "Before: "
   print *,insort
   CALL qsort(insort,array_len,array_size,cmp_function)
   print *, 'After: '
   print *, insort
END
!
   integer(2) function cmp_function(a1, a2)
   integer(2) a1, a2
   cmp_function=a1-a2
   end function
```

RADIX

Inquiry Intrinsic Function (Generic): Returns the base of the model representing numbers of

the same type and kind as the argument.

Syntax

result = **RADIX** (*x*)

x
(Input) 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 Model for Integer Data). For a real argument, the result has the value b (as defined in Model for Real Data).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DIGITS, EXPONENT, FRACTION, Data Representation Models

Example

If X is a REAL(4) value, RADIX (X) has the value 2.

RAISEQQ

Run-Time Function: Sends a signal to the executing program.

Module: USE DFLIB

Syntax

result = **RAISEQQ** (*sig*)

sig
(Input) INTEGER(4). Signal to raise. One of the following constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **SIG$ABORT**: Abnormal termination
- **SIG$FPE**: Floating-point error
- **SIG$ILL**: Illegal instruction
- **SIG$INT**: CTRL+C signal
- **SIG$SEGV**: Illegal storage access
- **SIG$TERM**: Termination request

If you do not install a signal handler (with **SIGNALQQ**, for example), when a signal occurs the system by default terminates the program with exit code 3.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

If a signal-handling routine for *sig* has been installed by a prior call to **SIGNALQQ**, **RAISEQQ** causes that routine to be executed. If no handler routine has been installed, the system terminates the program (the default action).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SIGNALQQ, SIGNAL, KILL

RAN

RAN can be used as an intrinsic function or as a run-time routine.

RAN Intrinsic Function

Nonelemental Intrinsic Function (Specific): Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1. **RAN** is not a pure function, so it cannot be referenced inside a **FORALL** construct.

Syntax

result = **RAN** (*i*)

i
(Input) 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 type is 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM, RANDOM_NUMBER

Example

In RAN (I), if variable I has the value 3, RAN has the value 4.8220158E-05.

RAN Run-Time Routine

Run-Time Function: Returns a pseudorandom number greater than or equal to zero and less than one from the uniform distribution.

Syntax

result = **RAN** (*iseed*)

iseed
(Input) INTEGER(4). Seed for the random number generator.

Results:

The result type is REAL(4). The result is a pseudorandom number, *x*, where 0 <= *x* < 1.

To ensure different random values for each run of a program, use different initial values of *iseed* (for example, use a reading from the system clock). The argument *iseed* should initially be set to a large, odd integer value. **RAN** stores a value in the argument *iseed* that it later uses to calculate the next random number.

The procedures **RANDOM**, **RAN**, and **RANDOM_NUMBER**, and the portability functions **DRAND**, **DRANDM**, **RAND**, **IRANDM**, **RAND**, and **RANDOM** use the same algorithms and thus return the same answers. They are all compatible and can be used interchangeably. (The algortihm used is a "Prime Modulus M Multiplicative Linear Congruential Generator," a modified version of the random number generator by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find," CACM, October 1988, Vol. 31, No. 10.)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM, RANDOM_NUMBER

Example

```
INTEGER(4) iseed
REAL(4) rnd
iseed = 425001
rnd = RAN(iseed)
```

RAND, RANDOM

Portability Functions: Return real random numbers in the range 0.0 through 1.0.

Module: USE DFLIB

Syntax

result = **RAND** ([*iflag*])
result = **RANDOM** (*iflag*)

iflag
(Input) INTEGER(4). Optional for **RAND**. Controls the way the random number is selected.

Results:

The result type is REAL(4). **RAND** and **RANDOM** return random numbers in the range 0.0 through 1.0.

Value of *iflag*	**Selection process**
1	The generator is restarted and the first random value is selected.
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using iflag, restarted, and the first random value is selected.

When **RAND** is called without an argument, *iflag* is assumed to be 0.

There is no difference between **RAND** and **RANDOM**. Both functions are included to ensure portability of existing code that references one or both of them. The intrinsic functions **RANDOM_NUMBER** and **RANDOM_SEED** provide the same functionality.

Note: Because Visual Fortran offers an intrinsic subroutine also called **RANDOM** in the default library, the only way to access this portability function is with the **USE DFPORT** statement. Without it, you can only access the default subroutine.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_NUMBER, RANDOM_SEED, RANDOM

RANDOM

Run-Time Subroutine: Returns a pseudorandom number greater than or equal to zero and less than one from the uniform distribution.

Module: USE DFLIB

Syntax

CALL RANDOM (*ranval*)

ranval
(Output) REAL(4). Pseudorandom number, 0 <= *ranval* < 1, from the uniform distribution.

A given seed always produces the same sequence of values from **RANDOM**.

If **SEED** is not called before the first call to **RANDOM**, **RANDOM** begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) to **SEED** before the first call to **RANDOM**.

All the random procedures (**RANDOM**, **RAN**, and **RANDOM_NUMBER**, and the portability functions **DRAND**, **DRANDM**, **RAND**, **IRANDM**, **RAND**, and **RANDOM)** use the same algorithms and thus return the same answers. They are all compatible and can be used interchangeably. (The algorithm used is a "Prime Modulus M Multiplicative Linear Congruential Generator," a modified version of the random number generator by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find," CACM, October 1988, Vol. 31, No. 10.)

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_NUMBER, SEED, DRAND and DRANDM, IRAND and IRANDM, RAN, RAND

Example

```
USE DFLIB
REAL(4) ran

CALL SEED(1995)
CALL RANDOM(ran)
```

RANDOM_NUMBER

Intrinsic Subroutine: Returns one pseudorandom number or an array of such numbers.

Syntax

CALL RANDOM_NUMBER (*harvest*)

harvest
(Output) Must be of type real. It 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.

The seed for the pseudorandom number generator used by **RANDOM_NUMBER** can be set or queried with RANDOM_SEED. If **RANDOM_SEED** is not used, the processor sets the seed for **RANDOM_NUMBER** to a processor-dependent value.

The **RANDOM_NUMBER** generator uses two separate congruential generators together to produce a period of approximately 10**18, and produces real pseudorandom results with a uniform distribution in (0,1). It accepts two integer seeds, the first of which is reduced to the range [1, 2147483562]. The second seed is reduced to the range [1, 2147483398]. This means that the generator effectively uses two 31-bit seeds.

For more information on the algorithm, see the following:

- Communications of the ACM vol 31 num 6 June 1988, titled: Efficient and Portable Combined Random Number Generators by Pierre L'ecuyer.
- Springer-Verlag New York, N. Y. 2nd ed. 1987, titled: A Guide to Simulation by Bratley, P., Fox, B. L., and Schrage, L. E.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_SEED, RANDOM, SEED, DRAND and DRANDM, IRAND and IRANDM, RAN, RAND and RANDOM

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.

The following shows another example:

```
 REAL x, array1 (5, 5)
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(x)
 CALL RANDOM_NUMBER(array1)
```

Consider also the following:

```
program testrand
  intrinsic random_seed, random_number
  integer size, seed(2), gseed(2), hiseed(2), zseed(2)
  real harvest(10)
  data seed /123456789, 987654321/
  data hiseed /-1, -1/
  data zseed /0, 0/
  call random_seed(SIZE=size)
  print *,"size ",size
  call random_seed(PUT=hiseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"hiseed gseed", hiseed, gseed
  call random_seed(PUT=zseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"zseed gseed ", zseed, gseed
  call random_seed(PUT=seed(1:size))
  call random_seed(GET=gseed(1:size))
  call random_number(HARVEST=harvest)
  print *, "seed gseed ", seed, gseed
  print *, "harvest"
  print *, harvest
  call random_seed(GET=gseed(1:size))
  print *,"gseed after harvest ", gseed
end program testrand
```

RANDOM_SEED

Intrinsic Subroutine: Changes or queries the seed (starting point) for the pseudorandom number generator used by RANDOM_NUMBER.

Syntax

CALL RANDOM_SEED([*size*] [, *put*] [, *get*])

size
(Optional; output) Must be scalar and of type default integer. Number of integers the processor uses to hold the value of the seed.

put
(Optional; input) Must be a default integer array of rank one. It is used by the processor to reset the value of the seed.

get
(Optional; output) Must be a default integer array of rank one. It is set to the current value of the seed.

No more than one argument can be specified. Both *put* and *get* must be greater than or equal to the size of the array the processor uses to store the seed. You can determine this size by calling **RANDOM_SEED** with the *size* argument (see second example).

If no argument is specified, a random number based on the date and time is assigned to the seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM_NUMBER, SEED, SRAND

Examples

Consider the following:

```
CALL RANDOM_SEED                          ! Processor initializes 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
```

The following shows another example:

```
 INTEGER I
 INTEGER, ALLOCATABLE :: new (:), old(:)
 CALL RANDOM_SEED ( )  ! Processor reinitializes the seed
                       ! randomly from the date and time
 CALL RANDOM_SEED (SIZE = I)  ! I is set to the size of
                              ! the seed array
 ALLOCATE (new(I))
 ALLOCATE (old(I))
 CALL RANDOM_SEED (GET=old(1:I)) ! Gets the current seed
 WRITE(*,*) old
 new = 5
 CALL RANDOM_SEED (PUT=new(1:I)) ! Sets seed from array
                                 ! new
 END
```

RANDU

Intrinsic Subroutine: Computes a pseudorandom number as a single-precision value.

Syntax

CALL RANDU (*i1*, *i2*, *x*)

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.

Example

```
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.

RANGE

Inquiry Intrinsic Function (Generic): Returns the decimal exponent range in the model representing numbers with the same kind parameter as the argument.

Syntax

result = **RANGE** (*x*)

x
(Input) 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 Model for Integer Data.

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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: HUGE, TINY

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})

READ

Statement: Transfers input data from external sequential, direct-access, or internal records.

Syntax

Sequential

Formatted

READ (*eunit*, *format* [, *advance*] [, *size*] [, *iostat*] [, *err*] [, *end*] [, *eor*]) [*io-list*]
READ *form* [, *io-list*]

Formatted: List-Directed

READ (*eunit*, * [, *iostat*] [, *err*] [, *end*]) [*io-list*]
READ * [, *io-list*]

Formatted: Namelist

READ (*eunit*, *nml-group* [, *iostat*] [, *err*] [, *end*])
READ *nml*

Unformatted

READ (*eunit* [, *iostat*] [, *err*] [, *end*]) [*io-list*]

Direct-Access

Formatted

READ (*eunit*, *format*, *rec* [, *iostat*] [, *err*]) [*io-list*]

Unformatted

READ (*eunit*, *rec* [, *iostat*] [, *err*]) [*io-list*]

Indexed (VMS only)

Formatted

READ (*eunit, format, key* [,*keyid*] [,*iostat*] [,*err*]) [*io-list*]

Unformatted

READ (*eunit, key* [,*keyid*] [,*iostat*] [,*err*]) [*io-list*]

Internal

READ (*iunit, format* [, *iostat*] [, *err*] [, *end*]) [*io-list*]

eunit
Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.

format
Is a format specifier. It is optionally prefaced by FMT= if *format* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=.

For internal **READ**s, an asterisk (*) indicates list-directed formatting. For direct-access **READ**s, an asterisk is not permitted.

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 the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.

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: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-**DO** list.

form

Is the nonkeyword form of a format specifier (no FMT=).

*
Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml-group
Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier.

nml
Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist I/O.

rec
Is the cell number of a record to be accessed directly. Optionally prefaced by REC=.

key (VMS only)
Is a key specifier (KEY[*con*]=*value*).

keyid (VMS only)
Is a key-of-reference specifier (KEYID=*kn*).

iunit
Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list.

It must be a character variable. It must not be an array section with a vector subscript.

If a parameter of the **READ** statement is an expression that calls a function, that function must not execute an I/O statement or the **EOF** intrinsic function, because the results are unpredictable.

If I/O is to or from a formatted device, *io-list* cannot contain derived type variables, but it can contain components of derived types. If I/O is to a binary or unformatted device, *io-list* can contain either derived type components or a derived type variable.

The **READ** statement can disrupt the results of certain graphics text functions (such as **SETTEXTWINDOW**) that alter the location of the cursor. You can avoid the problem by getting keyboard input with the GETCHARQQ function and echoing the keystrokes to the screen using OUTTEXT. Alternatively, you can use SETTEXTPOSITION to control cursor location.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: I/O Lists, I/O Control List, Forms for Sequential READ Statements, Forms for Direct-Access READ Statements, Forms for Indexed READ Statements (VMS only), Forms and Rules for Internal READ Statements, PRINT, WRITE, I/O Formatting

Example

```
DIMENSION ia(10,20)
! Read in the bounds for the array.
! Then read in the array in nested implied-DO lists
! with input format of 8 columns of width 5 each.
READ (6, 990) il, jl, ((ia(i,j), j = 1, jl), i =1, il)
990 FORMAT (2I5, /, (8I5))

! Internal read gives a variable string-represented numbers
CHARACTER*12 str
str = '123456'
READ (str,'(i6)') i

! List-directed read uses no specified format
REAL x, y
INTEGER i, j
READ (*,*) x, y, i, j
```

REAL Directive

Compiler Directive: Specifies the default real kind.

Syntax

*c***DEC$ REAL**:{ 4 | 8 }

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

Rules and Behavior

The **REAL** directive selects a size of 4 or 8 bytes (KIND=4 or KIND=8) for default real numbers. When the directive is in 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 appear only 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. **REAL** 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 following form is also allowed: `!MS$REAL:{4 | 8}`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REAL, INTEGER Directive, General Compiler Directives

Example

```
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
```

REAL Function

Elemental Intrinsic Function (Generic): Converts a value to real type.

Syntax

result = **REAL** (*a* [, *kind*])

a
(Input) Must be of type integer, real, or complex.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result is real type. 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 [2, 3]	INTEGER(4)	REAL(4)
REAL [2]	INTEGER(4)	REAL(4)
FLOATK [4]	INTEGER(8)	REAL(4)
	REAL(4)	REAL(4)
SNGL [2, 5]	REAL(8)	REAL(4)

SNGLQ [6]	REAL(16)	REAL(4)
	COMPLEX(4)	REAL(4)
	COMPLEX(8)	REAL(8)

[1] These specific functions cannot be passed as actual arguments.
[2] The setting of compiler option /real_size can affect FLOAT, REAL, and SNGL.
[3] Or FLOATJ. For compatibility with older versions of Fortran, FLOAT can also be specified as a generic function.
[4] Alpha only
[5] For compatibility with older versions of Fortran, SNGL can also be specified as a generic function. The generic SNGL includes specific function REAL, which takes a REAL(4) argument and produces a REAL(4) result.
[6] VMS, U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DFLOAT, DREAL, DBLE

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.

REAL

Statement: Specifies the REAL data type.

Syntax

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

n
Is kind 4 or 8.

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**.

To change the default kind value, use the /real_size compiler option or the cDEC$ REAL:8 directive.

REAL(4) and **REAL***4 (single precision) are the same data type. **REAL**(8), **REAL***8**,** and **DOUBLE PRECISION** are the same data type.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DOUBLE PRECISION, REAL directive, Real Data Types, General Rules for Real Constants, REAL(4) Constants, REAL(8) or DOUBLE PRECISION Constants

Examples

Entity-oriented examples are:

```
MODULE DATDECLARE
   REAL (8), OPTIONAL :: testval=50.d0
   REAL, SAVE :: a(10), b(20,30)
   REAL, PARAMETER :: x = 100.

Attribute-oriented examples are:
MODULE DATDECLARE
   REAL (8) testval=50.d0
   REAL x, a(10), b(20,30)
   OPTIONAL testval
   SAVE a, b
   PARAMETER (x = 100.)
```

RECORD

Statement: Declares a record structure as an entity with a name.

Syntax

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**, **EQUIVALENCE**, or **NAMELIST** statements.

Records initially have undefined values unless you have defined their values in structure declarations.

STRUCTURE and **RECORD** constructs have been replaced by derived types, which should be used in writing new code. See Derived Type and TYPE.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Derived Type, MAP...END MAP, STRUCTURE...END STRUCTURE, TYPE, UNION...END UNION, Record Structures

Example

```
STRUCTURE /address/
  LOGICAL*2    house_or_apt
  INTEGER*2    apt
  INTEGER*2    housenumber
  CHARACTER*30 street
  CHARACTER*20 city
  CHARACTER*2  state
  INTEGER*4    zip
END STRUCTURE

RECORD /address/ mailing_addr(20), shipping_addr(20)
```

RECTANGLE, RECTANGLE_W

Graphics Function: Draws a rectangle using the current graphics color, logical write mode, and line style.

Module: USE DFLIB

Syntax

result = **RECTANGLE** (*control*, *x1*, *y1*, *x2*, *y2*)
result = **RECTANGLE_W** (*control*, *wx1*, *wy1*, *wx2*, *wy2*)

control
(Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GFILLINTERIOR**: Draws a solid figure using the current color and fill mask.
- **$GBORDER**: Draws the border of a rectangle using the current color and line style.

x1, y1
(Input) INTEGER(2). Viewport coordinates for upper-left corner of rectangle.

x2, y2
(Input) INTEGER(2). Viewport coordinates for lower-right corner of rectangle.

wx1, *wy1*
(Input) REAL(8). Window coordinates for upper-left corner of rectangle.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of rectangle.

Results:

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0.

The **RECTANGLE** function uses the viewport-coordinate system. The viewport coordinates (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle.

The **RECTANGLE_W** function uses the window-coordinate system. The window coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*) are the diagonally opposed corners of the rectangle.

SETCOLORRGB sets the current graphics color. **SETFILLMASK** sets the current fill mask. By default, filled graphic shapes are filled solid with the current color.

If you fill the rectangle using **FLOODFILLRGB**, the rectangle must be bordered by a solid line style. Line style is solid by default and can be changed with **SETLINESTYLE**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

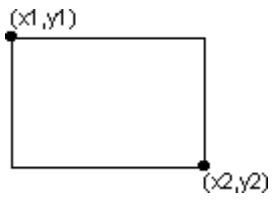
See Also: SETFILLMASK, GRSTATUS, LINETO, POLYGON, FLOODFILLRGB, SETLINESTYLE, SETCOLOR, SETWRITEMODE

Example

This program draws the rectangle shown below.

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(2) dummy, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
dummy = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
END
```

Figure: Output of Program RECTNGL.FOR



RECURSIVE

Keyword: Specifies that a subroutine or function can call itself directly or indirectly. Recursion is permitted if the keyword 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

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ENTRY, FUNCTION, SUBROUTINE, /recursive, OPTIONS, Program Units and Procedures

Example

```
! RECURS.F90
!
  i = 0
  CALL Inc (i)
  END

  RECURSIVE SUBROUTINE Inc (i)
  i = i + 1
  CALL Out (i)
  IF (i.LT.20) CALL Inc (i)    ! This also works in OUT
  END SUBROUTINE Inc

  SUBROUTINE Out (i)
```

```
    WRITE (*,*) i
    END SUBROUTINE Out
```

%REF

Built-in Function: Changes the form of an actual argument. Passes the argument by reference. In Visual Fortran, passing by reference is the default.

Syntax

result = **%REF** (*a*)

a
(Input) An expression, record name, procedure name, array, character array section, or array element.

You must specify **%REF** in the actual argument list of a **CALL** statement or function reference. You cannot use it in any other context.

The following table lists the DIGITAL Fortran defaults for argument passing, and the allowed uses of **%REF**:

Actual Argument Data Type	Default	%REF
Expressions:		
Logical	REF	Yes
Integer	REF	Yes
REAL(4)	REF	Yes
REAL(8)	REF	Yes
REAL(16) [1]	REF	Yes
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
Character	See table note [2]	Yes
Hollerith	REF	No
Aggregate [3]	REF	Yes
Derived	REF	Yes
Array Name:		
Numeric	REF	Yes

Character	See table note [2]	Yes
Aggregate [3]	REF	Yes
Derived	REF	Yes
Procedure Name:		
Numeric	REF	Yes
Character	See table note [2]	Yes

[1] VMS, U*X
[2] On DIGITAL UNIX, Windows NT and Windows 95 systems, a character argument is passed by address and hidden length.
[3] In DIGITAL Fortran record structures

See Also: CALL, %VAL

Example

```
CHARACTER(LEN=10) A, B
CALL SUB(A, %REF(B))
```

Variable A is passed by address and hidden length. Variable B is passed by reference.

REGISTERMOUSEEVENT

QuickWin Function: Registers the application-supplied callback routine to be called when a specified mouse event occurs in a specified window.

Module: USE DFLIB

Syntax

result = **REGISTERMOUSEEVENT** (*unit, mouseevents, callbackroutine*)

unit
(Input) INTEGER(4). Unit number of the window whose callback routine on mouse events is to be registered.

mouseevents
(Input) INTEGER(4). One or more mouse events to be handled by the callback routine to be registered. Symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) for the possible mouse events are:

- **MOUSE$LBUTTONDOWN**: Left mouse button down
- **MOUSE$LBUTTONUP**: Left mouse button up
- **MOUSE$LBUTTONDBLCLK**: Left mouse button double-click
- **MOUSE$RBUTTONDOWN**: Right mouse button down
- **MOUSE$RBUTTONUP**: Right mouse button up
- **MOUSE$RBUTTONDBLCLK**: Right mouse button double-click
- **MOUSE$MOVE**: Mouse moved

callbackroutine
(Input) EXTERNAL. Routine to be called on specified mouse event in the specified window. For a prototype mouse callback routine, see Using QuickWin.

Results:

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

- **MOUSE$BADUNIT**: The unit specified is not open, or is not associated with a QuickWin window.
- **MOUSE$BADEVENT**: The event specified is not supported.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, UNREGISTERMOUSEEVENT, WAITONMOUSEEVENT.

REMAPALLPALETTERGB, REMAPPALETTERGB

Graphics Function: REMAPALLPALETTERGB remaps a set of Red-Green-Blue (RGB) color values to indexes recognized by the video hardware. **REMAPPALETTERGB** remaps one color index to an RGB color value.

Module: USE DFLIB

Syntax

result = **REMAPALLPALETTERGB** (*colors*)
result = **REMAPPALETTERGB** (*index*, *color*)

colors

(Input) INTEGER(4). Ordered array of RGB color values to be mapped in order to indexes. Must hold 0-255 elements.

color
(Input) INTEGER(4). RGB color value to assign to a color index.

index
(Input) INTEGER(4). Color index to be reassigned an RGB color.

Results:

The result type is INTEGER(4). **REMAPALLPALETTERGB** returns 0 if successful; otherwise, -1. **REMAPPALETTERGB** returns the previous color assigned to the index.

The **REMAPALLPALETTERGB** function remaps all of the available color indexes simultaneously (up to 236; 20 indexes are reserved by the operating system). The *colors* argument points to an array of RGB color values. The default mapping between the first 16 indexes and color values is shown in the following table. The 16 default colors are provided with symbolic constants in DFLIB.F90 (in the \DF98\INCLUDE subdirectory).

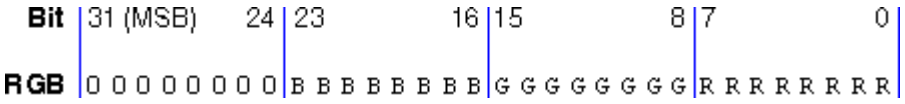
Index	Color	Index	Color
0	$BLACK	8	$GRAY
1	$BLUE	9	$LIGHTBLUE
2	$GREEN	10	$LIGHTGREEN
3	$CYAN	11	$LIGHTCYAN
4	$RED	12	$LIGHTRED
5	$MAGENTA	13	$LIGHTMAGENTA
6	$BROWN	14	$YELLOW
7	$WHITE	15	$BRIGHTWHITE

The number of colors mapped can be fewer than 236 if the number of colors supported by the current video mode is fewer, but at most 236 colors can be mapped by **REMAPALLPALETTERGB**. Most Windows graphics drivers support a palette of 256K colors or more, of which only a few can be mapped into the 236 palette indexes at a time. To access and use all colors on the system, bypass the palette and use direct RGB color functions such as such as **SETCOLORRGB** and **SETPIXELSRGB**.

Any RGB colors can be mapped into the 236 palette indexes. Thus, you could specify a palette with 236 shades of red. For further details on using different color procedures see Adding Color in the *Programmer's Guide*.

In each RGB color value, each of the three colors, red, green and blue, is represented by an eight-bit

value (2 hex digits). In the values you specify with **REMAPALLPALETTERGB** or **REMAPPALETTERGB,** red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, #008080 yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETBKCOLORRGB, SETCOLORRGB, SETBKCOLOR, SETCOLOR

Example

```
! Build as QuickWin or Standard Graphics App.

USE DFLIB
INTEGER(4) colors(3)
INTEGER(2) status

colors(1) = #00FFFF ! yellow
colors(2) = #FFFFFF ! bright white
colors(3) = 0       ! black
status = REMAPALLPALETTERGB(colors)

status = REMAPPALETTERGB(INT2(47), #45A315)
END
```

RENAME

Portability Function: Renames a file.

Module: USE DFPORT

Syntax

result = **RENAME** (*from*, *to*)

from
(Input) Character*(*). Path of an existing file.

to
(Input) Character*(*). The new path.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code, such as:

- **EACCES**: File or directory specified by *to* could not be created (invalid path). This error is also returned if the drive specified is not currently connected to a device.
- **ENOENT**: File or path specified by *from* could not be found.
- **EXDEV**: Attempt to move a file to a different device.

If the file specified in *to* exists, **RENAME** deletes it first.

It is possible to rename a file to itself without error.

The paths may use forward (/) or backward (\) slashes as path separators and can include drive letters.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RENAMEFILEQQ

Example

```
use dfport
integer(4) istatus
character*12 old_name, new_name
print *, "Enter file to rename: "
read *, old_name
print *, "Enter new name: "
read *, new_name
ISTATUS = RENAME (old_name, new_name)
```

RENAMEFILEQQ

Run-Time Function: Renames a file.

Module: USE DFLIB

Syntax

result = **RENAMEFILEQQ** (*oldname*, *newname*)

oldname
(Input) Character*(*). Current name of the file to be renamed.

newname
(Input) Character*(*). New name of the file to be renamed.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can use **RENAMEFILEQQ** to move a file from one directory to another on the same drive by giving a different path in the *newname* parameter.

If the function fails, call **GETLASTERRORQQ** to determine the reason. One of the following errors can be returned:

- **ERR$ACCESS**: The file specified by *newname* already exists or could not be created (invalid path).
- **ERR$NOENT**: File or path specified by *oldname* not found.
- **ERR$XDEV**: Attempt to move a file to a different device.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FINDFILEQQ, RENAME

Example

```
USE DFLIB
INTEGER(4) len
CHARACTER(80) oldname, newname
LOGICAL(4) result

WRITE(*,'(A, \)') ' Enter old name: '
len = GETSTRQQ(oldname)
WRITE(*,'(A, \)') ' Enter new name: '
len = GETSTRQQ(newname)
result = RENAMEFILEQQ(oldname, newname)
END
```

REPEAT

Transformational Intrinsic Function (Generic): Concatenates several copies of a string.

Syntax

result = **REPEAT** (*string*, *ncopies*)

string
(Input) Must be scalar and of type character.

ncopies
(Input) Must be scalar and of type integer. It must not be negative.

Results:

The result is a scalar of type character and length *ncopies* x **LEN**(*string*). The kind parameter is the same as *string*. The value of the result is the concatenation of *ncopies* copies of *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SPREAD

Example

REPEAT ('S', 3) has the value SSS.

REPEAT ('ABC', 0) has the value of a zero-length string.

The following shows another example:

```
CHARACTER(6) str
str = REPEAT('HO', 3)  ! returns HOHOHO
```

RESHAPE

Transformational Intrinsic Function (Generic): Constructs an array with a different shape from the argument array.

Syntax

result = **RESHAPE** (*source*, *shape* [, *pad*] [, *order*])

source
(Input) 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
(Input) 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
(Optional; input) 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
(Optional; input) 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.

In standard Fortran array element order, the first dimension varies fastest. For example, element order in a two-dimensional array would be (1,1), (2,1), (3,1) and so on. In a three-dimensional array, each dimension having two elements, the array element order would be (1,1,1), (2, 1, 1), (1, 2, 1), (2, 2, 1), (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2).

RESHAPE can be used to reorder a Fortran array to match C array ordering before the array is passed from a Fortran to a C procedure.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PACK, SHAPE, TRANSPOSE

Examples

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 3/)) has the value

```
[ 3  5  7 ]
[ 4  6  8 ].
```

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value

```
[ 3  4  5  6 ]
[ 7  8  1  1 ].
```

The following shows another example:

```
INTEGER AR1( 2, 5)
REAL F(5,3,8)
REAL C(8,3,5)
AR1 = RESHAPE((/1,2,3,4,5,6/),(/2,5/),(/0,0/),(/2,1/))
! returns      1 2 3 4 5
!              6 0 0 0 0
!
! Change Fortran array order to C array order
C = RESHAPE(F, (/8,3,5/), ORDER = (/3, 2, 1/))
END
```

RESULT

Keyword: Specifies a name for a function result.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FUNCTION, ENTRY, RECURSIVE, Program Units and Procedures

Examples

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
```

The following shows another example:

```
 recursive function FindSame(Aindex,Last,Used) &
&  result(FindSameResult)
 type(card) Last
 integer Aindex, i
 logical matched, used(5)
 if( Aindex > 5 ) then
   FindSameResult = .true.
   return
 endif
 . . .
```

RETURN

Statement: Transfers control from a subprogram to the calling program unit.

Syntax

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 90 and Fortran 95.)

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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CALL, CASE

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.

The following shows another example:

```
      SUBROUTINE Loop
      CHARACTER in
 10   READ (*, '(A)') in
      IF (in .EQ. 'Y') RETURN
      GOTO 10
 !    RETURN implied by the following statement:
      END

 !The following example demonstrates alternate returns:
      CALL AltRet (i, *10, *20, *30)
      WRITE (*, *) 'normal return'
      GOTO 40
 10   WRITE (*, *) 'I = 10'
      GOTO 40
 20   WRITE (*, *) 'I = 20'
      GOTO 40
 30   WRITE (*, *) 'I = 30'
 40   CONTINUE
      END
      SUBROUTINE AltRet (i, *, *, *)
      IF (i .EQ. 10) RETURN 1
      IF (i .EQ. 20) RETURN 2
      IF (i .EQ. 30) RETURN 3
      END
```

In this example, RETURN 1 specifies the list's first alternate-return label, which is a symbol for the actual argument *10 in the **CALL** statement. RETURN 2 specifies the second alternate-return label, and RETURN 3 specifies the third alternate-return label.

REWIND

Statement: Positions a sequential file at the beginning of the file (the initial point). It takes one of the following forms:

Syntax

REWIND ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])
REWIND *io-unit*

io-unit
(Input) Is an external unit specifier.

label
Is the label of the branch target statement that receives control if an error occurs.

i-var
(Output)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 or append access.

A **REWIND** statement is not allowed for a file that is open for direct access unless the compiler option fpscomp:general is specified.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: OPEN, READ, WRITE, Data Transfer I/O Statements, Branch Specifiers

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.

The following shows another example:

```
WRITE (7, '(I10)') int
REWIND (7)
READ (7, '(I10)') int
```

REWRITE

Statement: Rewrites the current record.

Formatted

REWRITE (*eunit*, *format* [, *iostat*] [, *err*]) [*io-list*]

Unformatted

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 a file with direct access.

The current record is the last record accessed by a preceding, successful sequential, 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.

Example

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)
```

RGBTOINTEGER

QuickWin Function: Converts three integers specifying red, green, and blue color intensities into a four-byte RGB integer for use with RGB functions and subroutines.

Module: USE DFLIB

Syntax

result = **RGBTOINTEGER** (*red, green, blue*)

red
(Input) INTEGER(4). Intensity of the red component of the RGB color value. Only the lower 8 bits of *red* are used.

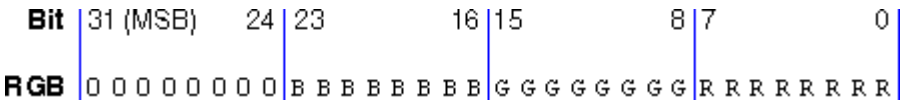
green
(Input) INTEGER(4). Intensity of the green component of the RGB color value. Only the lower 8 bits of *green* are used.

blue
(Input) INTEGER(4). Intensity of the blue component of the RGB color value. Only the lower 8 bits of *blue* are used.

Results:

The result type is INTEGER(4). The result is the combined RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value returned with **RGBTOINTEGER**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, INTEGERTORGB, SETCOLORRGB, SETBKCOLORRGB, SETPIXELRGB, SETPIXELSRGB, SETTEXTCOLORRGB.

Example

```
! Build as a QuickWin App.
USE DFLIB
INTEGER r, g, b, rgb, result
INTEGER(2) status
r = #F0
g = #F0
b = 0
rgb = RGBTOINTEGER(r, g, b)
result = SETCOLORRGB(rgb)
status = ELLIPSE($GFILLINTERIOR,INT2(40), INT2(55), &
                  INT2(90), INT2(85))
END
```

RINDEX

Portability Function: Locates the index of the last occurrence of a substring within a string.

Module: USE DFPORT

Syntax

result = **RINDEX** (*string*, *substr*)

string
(Input) Character*(*). Original string to search.

substr
(Input) Character*(*). String to search for.

Results:

The result type is INTEGER(4). The result is the starting position of the final occurence of *substrg* in *string*. Returns 0 if *substring* does not occur in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INDEX

Example

```
USE DFPORT
character*80 mainstring
character*4 shortstr
integer(4) where
mainstring="Hello Hello Hello Hello There There There"
shortstr="Hello"
where=rindex(mainstring,shortstr)
! where is 19
```

RRSPACING

Elemental Intrinsic Function (Generic): Returns the reciprocal of the relative spacing of model numbers near the argument value.

Syntax

result = **RRSPACING** (*x*)

x
(Input) 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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SPACING, Data Representation Models

Examples

If -3.0 is a REAL(4) value, RRSPACING (-3.0) has the value 0.75×2^{24}.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = RRSPACING(3.0)  ! returns 1.258291E+07
res4 = RRSPACING(-3.0) ! returns 1.258291E+07
r2 = 487923.3
res8 = RRSPACING(r2)   ! returns 8.382458680573952E+015
END
```

RSHIFT

Elemental Intrinsic Function: Shifts the bits in an integer right by a specified number of positions. For more information, see ISHFT.

RTC

Portability Function: Returns the number of seconds elapsed since a specific Greenwich mean time.

Module:USE DFPORT

Syntax

result = **RTC** ( )

Results:

The result type is REAL(8). The result is the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, TIME

Example

```
USE DFPORT
real(8) s, s1, time_spent
INTEGER(4) i, j
s = RTC( )
call sleep(4)
s1 = RTC( )
time_spent = s1 - s
PRINT *, 'It took ',time_spent, 'seconds to run.'
```

RUNQQ

Run-Time Function: Executes another program and waits for it to complete.

Module: USE DFLIB

Syntax

result = **RUNQQ** (*filename*, *commandline*)

filename
(Input) Character*(*). Filename of a program to be executed.

commandline
(Input) Character*(*). Command-line arguments passed to the program to be executed.

Results:

The result type is INTEGER(2). If the program executed with **RUNQQ** terminates normally, the exit code of that program is returned to the program that launched it. If the program fails, -1 is returned.

The **RUNQQ** function executes a new process for the operating system using the same path, environment, and resources as the process that launched it. The launching process is suspended until execution of the launched process is complete.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SYSTEM, NARGS

Example

See example in NARGS.

```
USE DFLIB
INTEGER(2) result
```

```
result = RUNQQ('myprog', '-c -r')
END
```

SAVE

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **SAVE** [, *att-ls*] **::** [*object* [, *object*]...]

Statement:

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 DIGITAL Fortran, the definitions of **COMMON** variables, and local variables of non- recursive subprograms (other than allocatable arrays or variables declared **AUTOMATIC**), are saved by default. To enhance portability and avoid possible compiler warning messages, DIGITAL 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):

- A blank common
- An object in a common block
- A procedure
- A dummy argument
- A function result
- An automatic object

- A **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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: COMMON, DATA, RECURSIVE, MOLULE, MODULE PROCEDURE, Type Declarations, Compatible attributes.

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
```

The following shows another example:

```
  SUBROUTINE MySub
    COMMON /z/ da, in, a, idum(10)
    real(8) x,y
    ...

    SAVE x, y, /z/
! alternate declaration
    REAL(8), SAVE :: x, y
    SAVE /z/
```

SAVEIMAGE, SAVEIMAGE_W

Graphics Function: Saves an image from a specified portion of the screen into a Windows bitmap file.

Module: USE DFLIB

Syntax

result = **SAVEIMAGE** (*filename*, *ulxcoord*, *ulycoord*, *lrxcoord*, *lrycoord*)
result = **SAVEIMAGE_W** (*filename*, *ulwxcoord*, *ulwycoord*, *lrwxcoord*, *lrwycoord*)

filename
(Input) Character*(*). Path of the bitmap file.

ulxcoord, *ulycoord*
(Input) INTEGER(4). Viewport coordinates for upper-left corner of the screen image to be captured.

lrxcoord, *lrycoord*
(Input) INTEGER(4). Viewport coordinates for lower-right corner of the screen image to be captured.

ulwxcoord, *ulwycoord*
(Input) REAL(8). Window coordinates for upper-left corner of the screen image to be captured.

lrwxcoord, *lrwycoord*
(Input) REAL(8). Window coordinates for lower-right corner of the screen image to be captured.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The **SAVEIMAGE** function captures the screen image within a rectangle defined by the upper-left and lower-right screen coordinates and stores the image as a Windows bitmap file specified by *filename*. The image is stored with a palette containing the colors displayed on the screen.

SAVEIMAGE defines the bounding rectangle in viewport coordinates. **SAVEIMAGE_W** defines the bounding rectangle in window coordinates.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETIMAGE, GETIMAGE_W, IMAGESIZE, IMAGESIZE_W, LOADIMAGE, LOADIMAGE_W, PUTIMAGE, PUTIMAGE_W

SCALE

Elemental Intrinsic Function (Generic): Returns the value of the exponent part (of the model for the argument) changed by a specified value.

Syntax

result = **SCALE** (*x*, *i*)

x
(Input) Must be of type real.

i
(Input) Must be of type integer.

Results:

The result type is the same as *x*. The result has the value *x* x b^i. Parameter *b* is defined in Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LSHIFT, Data Representation Models

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.

The following shows another example:

```
REAL r
r = SCALE(5.2, 2)     !returns 20.8
```

SCAN

Elemental Intrinsic Function (Generic): Scans a string for any character in a set of characters.

Syntax

result = **SCAN** (*string*, *set* [, *back*])

string
(Input) Must be of type character.

set
(Input) Must be of type character with the same kind parameter as *string*.

back
(Input) Must be of type logical.

Results:

The result type is default integer.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: VERIFY

Examples

SCAN (’ASTRING’, ’ST’) has the value 2.

SCAN (’ASTRING’, ’ST’, BACK=.TRUE.) has the value 3.

SCAN (’ASTRING’, ’CD’) has the value zero.

The following shows another example:

```
INTEGER i
INTEGER array(2)
i = SCAN (’FORTRAN’, ’TR’)                  ! returns 3
i = SCAN (’FORTRAN’, ’TR’, BACK = .TRUE.)  ! returns 5
i = SCAN (’FORTRAN’, ’GHA’)                 ! returns 6
i = SCAN (’FORTRAN’, ’ora’)                 ! returns 0
array = SCAN ((/’FORTRAN’,’VISUALC’/),(/’A’, ’A’/))
                                        ! returns (6, 5)
! Note that when using SCAN with arrays, the string
! elements must be the same length. When using string
! constants, blank pad to make strings the same length.
! For example:

array = SCAN ((/’FORTRAN’,’MASM ’/),(/’A’, ’A’/))
                                        ! returns (6, 2)
END
```

SCROLLTEXTWINDOW

Graphics Subroutine: Scrolls the contents of a text window.

Module: USE DFLIB

Syntax

CALL SCROLLTEXTWINDOW (*rows*)

rows
(Input) INTEGER(2). Number of rows to scroll.

The **SCROLLTEXTWINDOW** subroutine scrolls the text in a text window (previously defined by **SETTEXTWINDOW**). The default text window is the entire window.

The *rows* argument specifies the number of lines to scroll. A positive value for *rows* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling **CLEARSCREEN** ($GWINDOW). A value of 0 for *rows* has no effect.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: CLEARSCREEN, GETTEXTPOSITION, GETTEXTWINDOW, GRSTATUS, OUTTEXT, SETTEXTPOSITION, SETTEXTWINDOW, WRAPON

Example

```
!  Build as QuickWin or Standard Graphics app.
USE DFLIB
INTEGER(2) row
CHARACTER(18) string
TYPE (rccoord) oldpos

CALL SETTEXTWINDOW (INT2(1), INT2(0), &
                    INT2(25), INT2(80))
CALL CLEARSCREEN ( $GCLEARSCREEN )

DO row = 1, 6
   string = 'Hello, World # '
   CALL SETTEXTPOSITION( row, INT2(1), oldpos )
   WRITE(string(15:16), '(I2)') row
   CALL OUTTEXT( string )
END DO
WRITE(*,*) "Hit ENTER"
READ (*,*) ! wait for ENTER
!  Scroll window down 4 lines
CALL SCROLLTEXTWINDOW(INT2( -4) )
WRITE(*,*) "Hit ENTER"
READ( *,* ) ! wait for ENTER
!  Scroll window up 5 lines
CALL SCROLLTEXTWINDOW( INT2(5 ))
END
```

SCWRQQ (x86 only)

Run-Time Subroutine: Returns the floating-point processor control word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL SCWRQQ (*control*)

control
(Output) INTEGER(2). Floating-point processor control word.

SCRWQQ performs the same function as the run-time subroutine **GETCONTROLFPQQ**, and is provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LCWRQQ

Example

See the example in LCWRQQ.

SECNDS

SECNDS can be used as an intrinsic function or as a portability routine.

SECNDS Intrinsic Function

Elemental Intrinsic Function (Specific): Provides the system time of day, or elapsed time, as a floating-point value in seconds. **SECNDS** is not a pure function, so it cannot be referenced inside a **FORALL** construct.

result = **SECNDS** (*x*)

x
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, RTC, SYSTEM_CLOCK, TIME

Example

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
```

SECNDS Portability Routine

Portability Function: Returns the number of seconds that have elapsed since midnight, less the value of its argument.

Module: USE DFPORT

Syntax

result = **SECNDS** (*r*)

r
(Input) REAL(4). Number of seconds, precise to a hundredth of a second (0.01), to be subtracted.

Results:

The result type is REAL(4). The result is the number of seconds that have elapsed since midnight, minus *r*, with a precision of a hundredth of a second (0.01).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, RTC, SYSTEM_CLOCK, TIME

Example

```
USE DFPORT
REAL(4) s
INTEGER(4) i, j
s = SECNDS(0.0)
DO I = 1, 100000
   J = J + 1
END DO
s = SECNDS(s)
```

```
PRINT *, 'It took ',s, 'seconds to run.'
```

SEED

Run-Time Subroutine: Changes the starting point of the pseudorandom number generator.

Module: USE DFLIB

Syntax

CALL SEED (*iseed*)

iseed
(Input) INTEGER(4). Starting point for **RANDOM**.

SEED uses *iseed* to establish the starting point of the pseudorandom number generator. A given seed always produces the same sequence of values from **RANDOM**.

If **SEED** is not called before the first call to **RANDOM**, **RANDOM** always begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) to the **SEED** routine before the first call to **RANDOM**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RANDOM, RANDOM_SEED, RANDOM_NUMBER

Example

```
USE DFLIB
REAL rand
CALL SEED(7531)
CALL RANDOM(rand)
```

SELECT CASE...END SELECT

Statement: Transfers program control to a selected block of statements according to the value of a controlling expression. For more information, see CASE.

Example

```
CHARACTER*1 cmdchar
. . .
Files: SELECT CASE (cmdchar)
  CASE ('0')
    WRITE (*, *) "Must retrieve one to nine files"
  CASE ('1':'9')
    CALL RetrieveNumFiles (cmdchar)
  CASE ('A', 'a')
```

```
    CALL AddEntry
  CASE ('D', 'd')
    CALL DeleteEntry
  CASE ('H', 'h')
    CALL Help
  CASE DEFAULT
    WRITE (*, *) "Command not recognized; please re-enter"
END SELECT Files
```

SELECTED_INT_KIND

Transformational Intrinsic Function (Generic): Returns the value of the kind parameter of an integer data type.

Syntax

result = **SELECTED_INT_KIND** (*r*)

r
(Input) 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 more information, see Model for Integer Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SELECTED_REAL_KIND

Example

SELECTED_INT_KIND (6) = 4

The following shows another example:

```
i = SELECTED_INT_KIND(8)  ! returns 4
i = SELECTED_INT_KIND(3)  ! returns 2
i = SELECTED_INT_KIND(10) ! returns -1, precision
                          ! not available for this type
```

SELECTED_REAL_KIND

Transformational Intrinsic Function (Generic): Returns the value of the kind parameter of a real data type.

Syntax

result = **SELECTED_REAL_KIND**([*p*] [, *r*])

p
(Optional; input) Must be scalar and of type integer.

r
(Optional; input) 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 more information, see Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SELECTED_INT_KIND

Example

SELECTED_REAL_KIND (6, 70) = 8

The following shows another example:

```
i = SELECTED_REAL_KIND(r=200)  ! returns 8
i = SELECTED_REAL_KIND(13)     ! returns 8
```

SEQUENCE

Statement: Preserves the storage order of a derived-type definition.

Syntax

SEQUENCE

Rules and Behavior

The **SEQUENCE** statement allows derived types to be used in common blocks and to be equivalenced.

The **SEQUENCE** statement appears only as part of derived-type definitions. It causes the components of the derived type to be stored in the same sequence they are listed in the type definition. If you do not specify **SEQUENCE**, the physical storage order is not necessarily the same as the order of components in the type definition.

If a derived type is a sequence derived type, then any other derived type that includes it must also be a sequence type.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Derived Type, Data Types, Constants, and Variables

Example

```
!DEC$ PACK:1
TYPE NUM1_SEQ
  SEQUENCE
  INTEGER(2)::int_val
  REAL(4)::real_val
  LOGICAL(2)::log_val
END TYPE NUM1_SEQ
TYPE num2_seq
   SEQUENCE
   logical(2)::log_val
   integer(2)::int_val
   real(4)::real_val
end type num2_seq
type (num1_seq) num1
type (num2_seq) num2
character*8 t, t1
equivalence (num1,t)
equivalence (num2,t1)
num1%int_val=2
num1%real_val=3.5
num1%log_val=.TRUE.
t1(1:2)=t(7:8)
t1(3:4)=t(1:2)
t1(5:8)=t(3:6)
print *, num2%int_val, num2%real_val, num2%log_val
end
```

SETACTIVEQQ

QuickWin Function: Makes a child window active, but does *not* give it focus.

Module: USE DFLIB

Syntax

result = **SETACTIVEQQ** (*unit*)

unit
(Input) INTEGER(4). Unit number of the child window to be made active.

Results:

The result type is INTEGER(4). The result is 1 if successful; otherwise, 0.

When a window is made active, it receives graphics output (from **ARC**, **LINETO** and **OUTGTEXT**, for example) but is not brought to the foreground and does not have the focus. If a window needs to be brought to the foreground, it must be given the focus. A window is given focus with **FOCUSQQ**, by clicking it with the mouse, or by performing I/O other than graphics on it, unless the window was opened with IOFOCUS='.FALSE.'. By default, IOFOCUS='.TRUE.', except for child windows opened as unit '*'.

The window that has the focus is always on top, and all other windows have their title bars grayed out. A window can have the focus and yet not be active and not have graphics output directed to it. Graphical output is independent of focus.

If IOFOCUS='.TRUE.', the child window receives focus prior to each **READ, WRITE**, **PRINT**, or **OUTTEXT**. Calls to graphics functions (such as **OUTGTEXT** and **ARC**) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: GETACTIVEQQ, FOCUSQQ, INQFOCUSQQ, Using QuickWin

SETBKCOLOR

Graphics Function: Sets the current background color index for both text and graphics.

Module: USE DFLIB

Syntax

result = **SETBKCOLOR** (*color*)

color

(Input) INTEGER(4). Color index to set the background color to.

Results:

The result type is INTEGER(4). The result is the previous background color index.

SETBKCOLOR changes the background color index for both text and graphics. The color index of text over the background color is set with **SETTEXTCOLOR**. The color index of graphics over the background color (used by drawing functions such as **FLOODFILL** and **ELLIPSE**) is set with **SETCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use **SETBKCOLORRGB**, **SETCOLORRGB**, and **SETTEXTCOLORRGB**.

Changing the background color index does not change the screen immediately. The change becomes effective when **CLEARSCREEN** is executed or when doing text input or output, such as with **READ**, **WRITE**, or **OUTTEXT**. The graphics output function **OUTGTEXT** does not affect the color of the background.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are **GETBKCOLOR** and **SETBKCOLOR**. The default background color index is 0, which is associated with black unless the user remaps the palette with **REMAPPALETTERGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETBKCOLORRGB, GETBKCOLOR, REMAPALLPALETTERGB, REMAPPALETTERGB, SETCOLOR, SETTEXTCOLOR

Example

```
USE DFLIB
INTEGER(4) i
i = SETBKCOLOR(14)
```

SETBKCOLORRGB

Graphics Function: Sets the current background color to the given Red-Green-Blue (RGB) value.

Module: USE DFLIB

Syntax

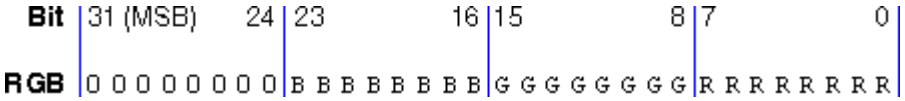
result = **SETBKCOLORRGB** (*color*)

color
(Input) INTEGER(4). RGB color value to set the background color to. Range and result depend on the system's display adapter.

Results:

The result type is INTEGER(4). The result is the previous background RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with **SETBKCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

The default background color is value 0, which is black. Changing the background color value does not change the screen immediately, but becomes effective when **CLEARSCREEN** is executed or when doing text input or output such as **READ**, **WRITE**, or **OUTTEXT**. The graphics output function **OUTGTEXT** does not affect the color of the background.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. The RGB color value of text over the background color (used by text functions such as **OUTTEXT**, **WRITE**, and **PRINT**) is set with **SETTEXTCOLORRGB**. The RGB color value of graphics over the background color (used by graphics functions such as **ARC**, **OUTGTEXT**, and **FLOODFILLRGB**) is set with **SETCOLORRGB**.

SETBKCOLORRGB (and the other RGB color selection functions **SETCOLORRGB**, and **SETTEXTCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (**SETCOLOR**, **SETBKCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETBKCOLORRGB, SETCOLORRGB, SETTEXTCOLORRGB, SETPIXELRGB, SETPIXELSRGB, SETBKCOLOR

Example

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(4) oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
```

```
x2 = 240; y2 = 150
oldcolor = SETBKCOLORRGB(#FF0000) !blue
oldcolor = SETCOLORRGB(#FF) ! red
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

SETCLIPRGN

Graphics Subroutine: Limits graphics output to part of the screen.

Module: USE DFLIB

Syntax

CALL SETCLIPRGN (*x1*, *y1*, *x2*, *x2*)

x1, *y1*
(Input) INTEGER(2). Physical coordinates for upper-left corner of clipping region.

x2, *y2*
(Input) INTEGER(2). Physical coordinates for lower-right corner of clipping region.

The **SETCLIPRGN** function limits the display of subsequent graphics output and font text output to that which fits within a designated area of the screen (the "clipping region"). The physical coordinates (*x1*, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangle that defines the clipping region. The **SETCLIPRGN** function does not change the viewport-coordinate system; it merely masks graphics output to the screen.

SETCLIPRGN affects graphics and font text output only, such as **OUTGTEXT**. To mask the screen for text output using **OUTTEXT**, use **SETTEXTWINDOW**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPHYSCOORD, GRSTATUS, SETTEXTWINDOW, SETVIEWORG, SETVIEWPORT, SETWINDOW

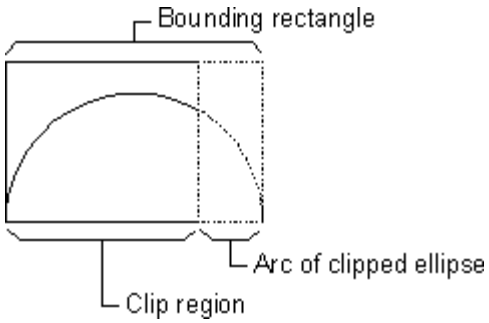
Example

This program draws an ellipse lying partly within a clipping region, as shown below.

```
! Build as QuickWin or Standard Graphics ap.
USE DFLIB
INTEGER(2) status, x1, y1, x2, y2
INTEGER(4) oldcolor
x1 = 10;  y1 = 50
x2 = 170; y2 = 150
! Draw full ellipse in white
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
oldcolor = SETCOLORRGB(#FF0000) !blue
```

```
WRITE(*,*) "Hit enter"
READ(*,*)
CALL CLEARSCREEN($GCLEARSCREEN) ! clear screen
CALL SETCLIPRGN( INT2(0), INT2(0), &
                 INT2(150), INT2(125))
! only part of ellipse inside clip region drawn now
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

Figure: Output of Program SETCLIP.FOR



SETCOLOR

Graphics Function: Sets the current graphics color index.

Module: USE DFLIB

Syntax

result = **SETCOLOR** (*color*)

color
(Input) INTEGER(2). Color index to set the current graphics color to.

Results:

The result type is INTEGER(2). The result is the previous color index if successful; otherwise, -1.

The **SETCOLOR** function sets the current graphics color index, which is used by graphics functions such as **ELLIPSE**. The background color index is set with **SETBKCOLOR**. The color index of text over the background color is set with **SETTEXTCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use **SETCOLORRGB**, **SETBKCOLORRGB**, and **SETTEXTCOLORRGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETCOLORRGB, GETCOLOR, REMAPPALETTERGB, SETBKCOLOR, SETTEXTCOLOR, SETPIXEL, SETPIXELS

Example

```
USE DFLIB
INTEGER(2) color, oldcolor
LOGICAL status
TYPE (windowconfig) wc

status = GETWINDOWCONFIG(wc)
color = wc.numcolors - 1
oldcolor = SETCOLOR(color)
END
```

SETCOLORRGB

Graphics Function: Sets the current graphics color to the specified Red-Green-Blue (RGB) value.

Module: USE DFLIB

Syntax

result = **SETCOLORRGB** (*color*)

color
(Input) INTEGER(4). RGB color value to set the current graphics color to. Range and result depend on the system's display adapter.

Results:

The result type is INTEGER(4). The result is the previous RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with **SETCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Bit	31 (MSB) 24	23 16	15 8	7 0
RGB	0 0 0 0 0 0 0 0	B B B B B B B B	G G G G G G G G	R R R R R R R R

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETCOLORRGB sets the RGB color value of graphics over the background color, used by the following graphics functions: **ARC**, **ELLIPSE**, **FLOODFILL**, **LINETO**, **OUTGTEXT**, **PIE**, **POLYGON**, **RECTANGLE**, and **SETPIXEL**. **SETBKCOLORRGB** sets the RGB color value of the current background for both text and graphics. **SETTEXTCOLORRGB** sets the RGB color value of text over the background color (used by text functions such as **OUTTEXT**, **WRITE**, and **PRINT**).

SETCOLORRGB (and the other RGB color selection functions **SETBKCOLORRGB,** and **SETTEXTCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (**SETCOLOR**, **SETBKCOLOR**, and **SETTEXTCOLOR**) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETBKCOLORRGB, SETTEXTCOLORRGB, GETCOLORRGB, ARC, ELLIPSE, FLOODFILLRGB, SETCOLOR, LINETO, OUTGTEXT, PIE, POLYGON, RECTANGLE, REMAPPALETTERGB, SETPIXELRGB, SETPIXELSRGB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE DFLIB
INTEGER(2) numfonts
INTEGER(4) oldcolor
TYPE (xycoord) xy
numfonts = INITIALIZEFONTS( )
oldcolor = SETCOLORRGB(#0000FF)   ! red
oldcolor = SETBKCOLORRGB(#00FF00) ! green
CALL MOVETO(INT2(200), INT2(100), xy)
CALL OUTGTEXT("hello, world")
END
```

SETCONTROLFPQQ (x86 only)

Run-Time Subroutine: Sets the value of the floating-point processor control word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL SETCONTROLFPQQ (*controlword*)

controlword
(Input) INTEGER(2). Floating-point processor control word.

The floating-point control word specifies how various exception conditions are handled by the floating-point math coprocessor, sets the floating-point precision, and specifies the floating-point rounding mechanism used.

The DFLIB.F90 module file (in the \DF98\INCLUDE subdirectory) contains constants defined for the control word as follows:

Parameter name	Hex value	Description
FPCW$MCW_IC	#1000	**Infinity control mask**
FPCW$AFFINE	#1000	Affine infinity
FPCW$PROJECTIVE	#0000	Projective infinity
FPCW$MCW_PC	#0300	**Precision control mask**
FPCW$64	#0300	64-bit precision
FPCW$53	#0200	53-bit precision
FPCW$24	#0000	24-bit precision
FPCW$MCW_RC	#0C00	**Rounding control mask**
FPCW$CHOP	#0C00	Truncate
FPCW$UP	#0800	Round up
FPCW$DOWN	#0400	Round down
FPCW$NEAR	#0000	Round to nearest
FPCW$MSW_EM	#003F	**Exception mask**
FPCW$INVALID	#0001	Allow invalid numbers
FPCW$DENORMAL	#0002	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	#0004	Allow divide by zero
FPCW$OVERFLOW	#0008	Allow overflow
FPCW$UNDERFLOW	#0010	Allow underflow
FPCW$INEXACT	#0020	Allow inexact precision

The defaults for the floating-point control word are 53-bit precision, round to nearest, and the denormal, underflow and inexact precision exceptions disabled. An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0.

Setting the floating-point precision and rounding mechanism can be useful if you are reusing old code that is sensitive to the floating-point precision standard used and you want to get the same results as on the old machine.

You can use **GETCONTROLFPQQ** to retrieve the current control word and **SETCONTROLFPQQ** to change the control word. Most users do not need to change the default settings. If you need to change the control word, always use **SETCONTROLFPQQ** to make sure

that special routines handling floating-point stack exceptions and abnormal propagation work correctly.

For a full discussion of the floating-point control word, exceptions, and error handling, see The Floating-Point Environment in the *Programmer's Guide*.

The Visual Fortran exception handler allows for software masking of invalid operations, but does not allow the math chip to mask them. If you choose to use the software masking, be aware that this can affect program performance if you compile code written for Visual Fortran with another compiler.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETCONTROLFPQQ, GETSTATUSFPQQ, LCWRQQ, SCWRQQ

Example

```
USE DFLIB
INTEGER(2) status, control, controlo

CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!     Save old control word
controlo = control
!     Clear all flags
control = control .AND. #0000
!     Set new control to round up
control = control .OR. FPCW$UP
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

SETDAT

Run-Time Function: Sets the system date.

Module: USE DFLIB

Syntax

result = **SETDAT** (*iyr*, *imon*, *iday*)

iyr
(Input) INTEGER(2). Year (*xxxx* AD).

imon
(Input) INTEGER(2). Month (1-12).

iday

(Input) INTEGER(2). Day of the month (1-31).

Results:

The result type is LOGICAL(4). The result is .TRUE. if the system date is changed; .FALSE. if no change is made.

Actual arguments of the function **SETDAT** can be any legal INTEGER(2) expression.

Refer to your operating system documentation for the range of permitted dates.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDAT, GETTIM, SETTIM

Example

```
USE DFLIB
LOGICAL(4) success
success = SETDAT(INT2(1997+1), INT2(2*3), INT2(30))
END
```

SETENVQQ

Run-Time Function: Sets the value of an existing environment variable, or adds and sets a new environment variable.

Module: USE DFLIB

Syntax

result = **SETENVQQ** (*varname=value*)

varname = value
(Input) Character*(*). String containing both the name and the value of the variable to be added or modified. Must be in the form: *varname = value*, where *varname* is the name of an environment variable and *value* is the value being assigned to it.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Environment variables define the environment in which a program executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

SETENVQQ deletes any terminating blanks in the string. Although the equal sign (=) is an illegal character within an environment value, you can use it to terminate *value* so that trailing blanks are

preserved. For example, the string PATH= =*value*'.

You can use **SETENVQQ** to remove an existing variable by giving a variable name followed by an equal sign with no value. For example, LIB= removes the variable LIB from the list of environment variables. If you specify a value for a variable that already exists, its value is changed. If the variable does not exist, it is created.

SETENVQQ affects only the environment that is local to the current process. You cannot use it to modify the command-level environment. When the current process terminates, the environment reverts to the level of the parent process. In most cases, this is the operating system level. However, you can pass the environment modified by **SETENVQQ** to any child process created by **RUNQQ**. These child processes get new variables and/or values added by **SETENVQQ**.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETENVQQ

Example

```
USE DFLIB
LOGICAL(4) success
success = SETENVQQ("PATH=c:\mydir\tmp")
success = &
SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")
END
```

SETERRORMODEQQ

Run-Time Subroutine: Sets the prompt mode for critical errors that by default generate system prompts.

Module: USE DFLIB

Syntax

CALL SETERRORMODEQQ (*pmode*)

pmode
(Input) LOGICAL(4). Flag that determines whether a prompt is displayed when a critical error occurs.

Certain I/O errors cause the system to display an error prompt. For example, attempting to write to a disk drive with the drive door open generates an "Abort, Retry, Ignore" message. When the system starts up, system error prompting is enabled by default (*pmode* = .TRUE.). You can also enable system error prompts by calling **SETERRORMODEQQ** with *pmode* set to ERR$HARDPROMPT (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory).

If prompt mode is turned off, a critical error that by default causes a system prompt will not cause a

system prompt. Erroneous I/O statements such as **OPEN**, **READ**, and **WRITE** fail immediately instead of being interrupted with prompts. This allows you to intercept failures in the I/O statement (by setting ERR=*errlabel*, for example, where *errlabel* designates an executable statement) and to take a different action than that requested by the system prompt, such as opening a temporary file, giving a more informative error message, or exiting. You can turn off prompt mode by setting *pmode* to .FALSE. or to the constant ERR$HARDFAIL (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory).

Note that **SETERRORMODEQQ** affects only errors that generate a system prompt. It does not affect other I/O errors, such as writing to a nonexistent file or attempting to open a nonexistent file with STATUS='OLD'.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!PROGRAM 1
!  DRIVE B door open
OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Generates a system prompt error here and waits for the user
!  to respond to the prompt before continuing
100   WRITE(*,*) ' Continuing'
END

! PROGRAM 2
!  DRIVE B door open
 USE DFLIB
 CALL SETERRORMODEQQ(.FALSE.)
 OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Causes the statement at label 100 to execute
!  without system prompt
100   WRITE(*,*) ' Drive B: not available, opening      &
                  &alternative drive.'
 OPEN (10, FILE = 'C:\NOFILE.DAT')
 END
```

SETEXITQQ

QuickWin Function: Sets a QuickWin application's exit behavior.

Module: USE DFLIB

Syntax

result = **SETEXITQQ** (*exitmode*)

exitmode
(Input) INTEGER(4). Determines the program exit behavior. The following exit parameters are defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory):

- **QWIN$EXITPROMPT**: Displays the following message box:

"Program exited with exit status X. Exit Window?"
where X is the exit status from the program. If Yes is entered, the application closes the window and terminates. If No is entered, the dialog box disappears and you can manipulate the windows as usual. You must then close the window manually.

- **QWIN$EXITNOPERSIST**: Terminates the application without displaying a message box.
- **QWIN$EXITPERSIST**: Leaves the application open without displaying a message box.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The default for both QuickWin and Standard Graphics applications is QWIN$EXITPROMPT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETEXITQQ, Using QuickWin

Example

```
! Build as QuickWin Ap
USE DFLIB
INTEGER(4) exmode, result

WRITE(*,'(1X,A,/)') 'Please enter the exit mode 1, 2   &
                     or 3 '
READ(*,*) exmode
SELECT CASE (exmode)
  CASE (1)
    result = SETEXITQQ(QWIN$EXITPROMPT)
  CASE (2)
    result = SETEXITQQ(QWIN$EXITNOPERSIST)
  CASE (3)
    result = SETEXITQQ(QWIN$EXITPERSIST)
  CASE DEFAULT
    WRITE(*,*) 'Invalid option - checking for bad     &
                return'
    IF(SETEXITQQ( exmode ) .NE. -1) THEN
      WRITE(*,*) 'Error not returned'
    ELSE
      WRITE(*,*) 'Error code returned'
    ENDIF
END SELECT
END
```

SET_EXPONENT

Elemental Intrinsic Function (Generic): Returns the value of the exponent part (of the model for the argument) set to a specified value.

Syntax

result = **SET_EXPONENT** (*x*, *i*)

x
(Input) Must be of type real.

i
(Input) Must be of type integer.

Results:

The result type is the same as *x*. The result has the value *x* x b^{i-e}. Parameters *b* and *e* are defined in Model for Real Data. If *x* has the value zero, the result is zero.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EXPONENT, Data Representation Models

Example

If 3.0 is a REAL(4) value, SET_EXPONENT (3.0, 1) has the value 1.5.

SETFILEACCESSQQ

Run-Time Function: Sets the file access mode for a specified file.

Module: USE DFLIB

Syntax

result = **SETFILEACCESSQQ** (*filename*, *access*)

filename
(Input) Character*(*). Name of a file to set access for.

access
(Input) INTEGER(4). Constant that sets the access. Can be any combination of the following flags, combined by an inclusive OR (such as IOR or OR):

- **FILE$ARCHIVE**: Marked as having been copied to a backup device.
- **FILE$HIDDEN**: Hidden. The file does not appear in the directory list that you can request from the command console.
- **FILE$NORMAL**: No special attributes (default).
- **FILE$READONLY**: Write-protected. You can read the file, but you cannot make changes to it.
- **FILE$SYSTEM**: Used by the operating system.

The flags are defined in module DFLIB.F90 in the \DF98\INCLUDE subdirectory.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

To set the access value for a file, add the constants representing the appropriate access.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETFILEINFOQQ

Example

```
USE DFLIB
INTEGER(4) permit
LOGICAL(4) result

permit = 0   ! clear permit
permit = FILE$READONLY + FILE$HIDDEN
result = SETFILEACCESSQQ ('formula.f90', permit)
END
```

SETFILETIMEQQ

Run-Time Function: Sets the modification time for a specified file.

Module: USE DFLIB

Syntax

result = **SETFILETIMEQQ** (*filename*, *timedate*)

filename
(Input) Character*(*). Name of a file.

timedate
(Input) INTEGER(4). Time and date information, as packed by **PACKTIMEQQ**.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The modification time is the time the file was last modified and is useful for keeping track of different versions of the file. The process that calls **SETFILETIMEQQ** must have write access to the file; otherwise, the time cannot be changed. If you set *timedate* to FILE$CURTIME (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory), **SETFILETIMEQQ** sets the modification time to

the current system time.

If the function fails, call **GETLASTERRORQQ** to determine the reason, which can be one of the following:

Error	Meaning
ERR$ACCESS	The path specifies a directory or a read-only file.
ERR$INVAL	Invalid argument; the *timedate* argument is invalid.
ERR$MFILE	Too many open files (the file must be opened to change its modification time).
ERR$NOENT	File or path not found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PACKTIMEQQ, UNPACKTIMEQQ, GETLASTERRORQQ

Example

```
USE DFLIB
INTEGER(2) day, month, year
INTEGER(2) hour, minute, second, hund
INTEGER(4) timedate
LOGICAL(4) result

CALL GETDAT(year, month, day)
CALL GETTIM(hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day,    &
                 hour, minute, second)
result = SETFILETIMEQQ('myfile.dat', timedate)
END
```

SETFILLMASK

Graphics Subroutine: Sets the current fill mask to a new pattern.

Module: USE DFLIB

Syntax

CALL SETFILLMASK (*mask*)

mask
(Input) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element

(byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of zero are set to the current background color. The current graphics color is set with **SETCOLORRGB** or **SETCOLOR**. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with **SETFILLMASK**), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (**FLOODFILLRGB**, **PIE**, **ELLIPSE**, **POLYGON**, and **RECTANGLE**).

To change the current fill mask, determine the array of bytes that corresponds to the desired bit pattern and set the pattern with **SETFILLMASK**, as in the following example.

Bit pattern								Value in mask
	●	○	○	●	○	○	●	●	mask(1) = #93
	●	●	○	○	●	○	○	●	mask(2) = #C9
	○	●	●	○	○	●	○	○	mask(3) = #64
	●	○	●	●	○	○	●	○	mask(4) = #B2
	○	●	○	●	●	○	○	●	mask(5) = #59
	○	○	●	○	●	●	○	○	mask(6) = #2C
	●	○	○	●	○	●	●	○	mask(7) = #96
	○	●	○	○	●	○	●	●	mask(8) = #4B
bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: ELLIPSE, FLOODFILLRGB, GETFILLMASK, PIE, POLYGON, RECTANGLE

Example

This program draws six rectangles, each with a different fill mask, as shown below.

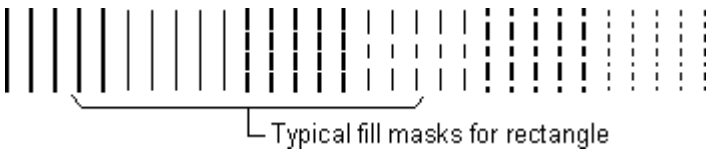
```
! Build as QuickWin or Standard Graphics Ap.
USE DFLIB

INTEGER(1), TARGET :: style1(8) &
            /#18,#18,#18,#18,#18,#18,#18,#18/
INTEGER(1), TARGET :: style2(8) &
            /#08,#08,#08,#08,#08,#08,#08,#08/
INTEGER(1), TARGET :: style3(8) &
            /#18,#00,#18,#18,#18,#00,#18,#18/
INTEGER(1), TARGET :: style4(8) &
            /#00,#08,#00,#08,#08,#08,#08,#08/
INTEGER(1), TARGET :: style5(8) &
            /#18,#18,#00,#18,#18,#00,#18,#18/
INTEGER(1), TARGET :: style6(8) &
            /#08,#00,#08,#00,#08,#00,#08,#00/
INTEGER(1) oldstyle(8) ! Placeholder for old style
INTEGER loop
INTEGER(1), POINTER :: ptr(:)
```

```
CALL GETFILLMASK( oldstyle )
! Make 6 rectangles, each with a different fill
DO loop = 1, 6
  SELECT CASE (loop)
    CASE (1)
      ptr => style1
    CASE (2)
      ptr => style2
    CASE (3)
      ptr => style3
    CASE (4)
      ptr => style4
    CASE (5)
      ptr => style5
    CASE (6)
      ptr => style6
  END SELECT
  CALL SETFILLMASK( ptr)
  status = RECTANGLE($GFILLINTERIOR,INT2(loop*40+5), &
               INT2(90),INT2((loop+1)*40), INT2(110))
END DO

CALL SETFILLMASK( oldstyle )  ! Restore old style
READ (*,*)                    ! Wait for ENTER to be
                              ! pressed
END
```

Figure: Output of Program SETFILL.FOR



SETFONT

Graphics Function: Finds a single font that matches a specified set of characteristics and makes it the current font used by the **OUTGTEXT** function.

Module: USE DFLIB

Syntax

result = **SETFONT** (*options*)

options
(Input) Character*(*). String describing font characteristics (see below for details).

Results:

The result type is INTEGER(2). The result is the index number (*x* as used in the **n***x* option) of the font if successful; otherwise, -1.

The **SETFONT** function searches the list of available fonts for a font matching the characteristics

specified in *options*. If a font matching the characteristics is found, it becomes the current font. The current font is used in all subsequent calls to the **OUTGTEXT** function. There can be only one current font.

The *options* argument consists of letter codes, as follows, that describe the desired font. The *options* parameter is not case sensitive or position sensitive.

t'*fontname*'	Name of the desired typeface. It can be any installed font.
h*y*	Character height, where *y* is the number of pixels.
w*x*	Select character width, where *x* is the number of pixels.
f	Select only a fixed-space font (do not use with the **p** characteristic).
p	Select only a proportional-space font (do not use with the **f** characteristic).
v	Select only a vector-mapped font (do not use with the **r** characteristic). In Windows NT, Roman, Modern, and Script are examples of vector-mapped fonts, also called plotter fonts. True Type fonts (for example, Arial, Symbol, and Times New Roman) are not vector-mapped.
r	Select only a raster-mapped (bitmapped) font (do not use with the **v** characteristic). In Windows NT, Courier, Helvetica, and Palatino are examples of raster-mapped fonts, also called screen fonts. True Type fonts are not raster-mapped.
e	Select the bold text format. This parameter is ignored if the font does not allow the bold format.
u	Select the underline text format. This parameter is ignored if the font does not allow underlining.
i	Select the italic text format. This parameter is ignored if the font does not allow italics.
b	Select the font that best fits the other parameters specified.
n*x*	Select font number *x*, where *x* is less than or equal to the value returned by the **INTIALIZEFONTS** function.

You can specify as many options as you want, except with **n***x*, which should be used alone. If you specify options that are mutually exclusive (such as the pairs **f/p** or **r/v**), the **SETFONT** function ignores them. There is no error detection for incompatible parameters used with **n***x*.

If the **b** option is specified and at least one font is initialized, **SETFONT** sets a font and returns 0 to indicate success.

In selecting a font, the **SETFONT** routine uses the following criteria, rated from highest precedence to lowest:

1. Pixel height
2. Typeface
3. Pixel width
4. Fixed or proportional font

You can also specify a pixel width and height for fonts. If you choose a nonexistent value for either and specify the **b** option, **SETFONT** chooses the closest match.

A smaller font size has precedence over a larger size. If you request Arial 12 with best fit, and only Arial 10 and Arial 14 are available, **SETFONT** selects Arial 10.

If you choose a nonexistent value for pixel height and width, the **SETFONT** function applies a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if you specify the **r** option (raster-mapped font), or if you request a specific typeface and do not specify the **b** option (best-fit).

If you specify the **n***x* parameter, **SETFONT** ignores any other specified options and supplies only the font number corresponding to *x*.

If a height is given, but not a width, or vice versa, **SETFONT** computes the missing value to preserve the correct font proportions.

The font functions affect only **OUTGTEXT** and the current graphics position; no other Fortran Graphics Library output functions are affected by font usage.

For each window you open, you must call **INITIALIZEFONTS** before calling **SETFONT**. **INITIALIZEFONTS** needs to be executed after each new child window is opened in order for a subsequent **SETFONT** call to be successful.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETFONTINFO, GETGTEXTEXTENT, GRSTATUS, OUTGTEXT, INITIALIZEFONTS, SETGTEXTROTATION

Example

```
!  Build as a Graphics ap.
USE DFLIB
INTEGER(2) fontnum, numfonts
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
!  Set typeface to Arial, character height to 18,
!  character width to 10, and italic
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Demo text')
END
```

SETGTEXTROTATION

Graphics Subroutine: Sets the orientation angle of the font text output in degrees. The current orientation is used in calls to **OUTGTEXT**.

Module: USE DFLIB

Syntax

CALL SETGTEXTROTATION (*degrees*)

degrees
(Input) INTEGER(4). Angle of orientation, in tenths of degrees, of the font text output.

The orientation of the font text output is set in tenths of degrees. Horizontal is 0°, and angles increase counterclockwise so that 900 (90°) is straight up, 1800 (180°) is upside down and left, 2700 (270°) is straight down, and so forth. If the user specifies a value greater than 3600 (360°), the subroutine takes a value equal to:

MODULO (user-specified tenths of degrees, 3600)

Although **SETGTEXTROTATION** accepts arguments in tenths of degrees, only increments of one full degree differ visually from each other on the screen.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETGTEXTROTATION

Example

```
! Build as a Graphics ap.
USE DFLIB
INTEGER(2) fontnum, numfonts
INTEGER(4) oldcolor, deg
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Straight text')
deg = -1370
CALL SETGTEXTROTATION(deg)
oldcolor = SETCOLORRGB(#008080)
CALL OUTGTEXT('Slanted text')
END
```

SETLINESTYLE

Graphics Subroutine: Sets the current line style to a new line style.

Module: USE DFLIB

Syntax

CALL SETLINESTYLE (*mask*)

mask
(Input) INTEGER(2). Desired Quickwin line-style mask. (See the table below.)

The mask is mapped to the style that most closly equivalences the the percentage of the bits in the mask that are set. The style produces lines that cover a certain percentage of the pixels in that line.

SETLINESTYLE sets the style used in drawing a line. You can choose from the following styles:

QuickWin Mask	Internal Windows Style	Selection Criteria	Appearance
0xFFFF	PS_SOLID	16 bits on	_____
0xEEEE	PS_DASH	11 to 15 bits on	----------
0xECEC	PS_DASHDOT	10 bits on	-.-.-.-.-.-.-.-.-.-
0xECCC	PS_DASHDOTDOT	9 bits on	-..-..-..-..-..-..-..
0xAAAA	PS_DOT	1 to 8 bits on	.....
0x0000	PS_NULL	0 bits on	

SETLINESTYLE affects the drawing of straight lines as in **LINETO**, **POLYGON**, and **RECTANGLE**, but not the drawing of curved lines as in **ARC**, **ELLIPSE**, or **PIE**.

The current graphics color is set with **SETCOLORRGB** or **SETCOLOR**. **SETWRITEMODE** affects how the line is displayed.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETLINESTYLE, GRSTATUS, LINETO, POLYGON, RECTANGLE, SETCOLOR, SETWRITEMODE

Example

```
! Build as a Graphics ap.
 USE DFLIB
 INTEGER(2)     status, style
 TYPE (xycoord) xy
```

```
style = #FFFF
CALL SETLINESTYLE(style)
CALL MOVETO(INT2(50), INT2(50), xy )
status = LINETO(INT2(300), INT2(300))
END
```

SETMESSAGEQQ

QuickWin Subroutine: Changes QuickWin status messages, state messages, and dialog box messages.

Module: USE DFLIB

Syntax

CALL SETMESSAGEQQ (*msg, id*)

msg
(Input) Character*(*). Message to be displayed. Must be a regular Fortran string, not a C string. Can include multibyte characters.

id
(Input) INTEGER(4). Identifier of the message to be changed. The following table shows the messages that can be changed and their identifiers:

Id	Message
QWIN$MSG_TERM	"Program terminated with exit code"
QWIN$MSG_EXITQ	"\nExit Window?"
QWIN$MSG_FINISHED	"Finished"
QWIN$MSG_PAUSED	"Paused"
QWIN$MSG_RUNNING	"Running"
QWIN$MSG_FILEOPENDLG	"Text Files(*.txt), *.txt; Data Files(*.dat), *.dat; All Files(*.*), *.*;"
QWIN$MSG_BMPSAVEDLG	"Bitmap Files(*.bmp), *.bmp; All Files(*.*), *.*;"
QWIN$MSG_INPUTPEND	"Input pending in"
QWIN$MSG_PASTEINPUTPEND	"Paste input pending"
QWIN$MSG_MOUSEINPUTPEND	"Mouse input pending in"
QWIN$MSG_SELECTTEXT	"Select Text in"
QWIN$MSG_SELECTGRAPHICS	"Select Graphics in"

QWIN$MSG_PRINTABORT	"Error! Printing Aborted."
QWIN$MSG_PRINTLOAD	"Error loading printer driver"
QWIN$MSG_PRINTNODEFAULT	"No Default Printer."
QWIN$MSG_PRINTDRIVER	"No Printer Driver."
QWIN$MSG_PRINTINGERROR	"Print: Printing Error."
QWIN$MSG_PRINTING	"Printing"
QWIN$MSG_PRINTCANCEL	"Cancel"
QWIN$MSG_PRINTINPROGRESS	"Printing in progress..."
QWIN$MSG_HELPNOTAVAIL	"Help Not Available for Menu Item"
QWIN$MSG_TITLETEXT	"Graphic"

Note that QWIN$MSG_FILEOPENDLG and QWIN$MSG_BMPSAVEDLG control the text in file choosing dialog boxes and have the following syntax:

"file description, file designation"

You can change any string produced by QuickWin by calling **SETMESSAGEQQ** with the appropriate *id*. This includes status messages displayed at the bottom of a QuickWin application, state messages (such as "Paused"), and dialog box messages. These messages can include multibyte characters. (For more information on multibyte characters, see Using National Language Support Routines in the *Programmer's Guide*.) To change menu messages, use **MODIFYMENUSTRINGQQ**.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: MODIFYMENUSTRINGQQ

Example

```
USE DFLIB
print*, "Hello"
CALL SETMESSAGEQQ('Changed exit text', QWIN$MSG_EXITQ)
```

SETPIXEL, SETPIXEL_W

Graphics Function: Sets a pixel at a specified location to the current graphics color index.

Module: USE DFLIB

Syntax

result = **SETPIXEL** (*x, y*)
result = **SETPIXEL_W** (*wx, wy*)

x, y
(Input) INTEGER(2). Viewport coordinates for target pixel.

wx, wy
(Input) REAL(8). Window coordinates for target pixel.

Results:

The result type is INTEGER(2). The result is the previous color index of the target pixel if successful; otherwise, -1 (for example, if the pixel lies outside the clipping region).

SETPIXEL sets the specified pixel to the current graphics color index. The current graphics color index is set with **SETCOLOR** and retrieved with **GETCOLOR**. The non-RGB color functions (such as **SETCOLOR** and **SETPIXELS**) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue(RGB) value with an RGB color function, rather than a palette index with a non-RGB color function. **SETPIXELRGB** and **SETPIXELRGB_W** give access to the full color capacity of the system by using direct color values rather than indexes to a palette.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETPIXELRGB, GETPIXEL, SETPIXELS, GETPIXELS, GETCOLOR, SETCOLOR

Example

```
! Build as a Graphics ap.
USE DFLIB
INTEGER(2) status, x, y
status = SETCOLOR(INT2(2))
x = 10
! Draw pixels.
DO y = 50, 389, 3
   status = SETPIXEL( x, y )
   x = x + 2
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

SETPIXELRGB, SETPIXELRGB_W

Graphics Function: Sets a pixel at a specified location to the specified Red-Green-Blue (RGB) color value.

Module: USE DFLIB

Syntax

result = **SETPIXELRGB** (*x*, *y*, *color*)
result = **SETPIXELRGB_W** (*wx*, *wy*, *color*)

x, *y*
(Input) INTEGER(2). Viewport coordinates for target pixel.

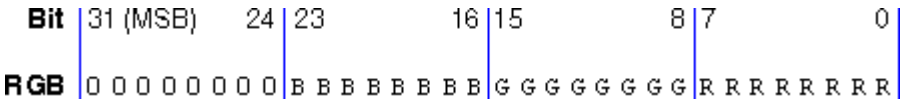
wx, *wy*
(Input) REAL(8). Window coordinates for target pixel.

color
(Input) INTEGER(4). RGB color value to set the pixel to. Range and result depend on the system's display adapter.

Results:

The result type is INTEGER(4). The result is the previous RGB color value of the pixel.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with **SETPIXELRGB** or **SETPIXELRGB_W**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

If any of the pixels are outside the clipping region, those pixels are ignored.

SETPIXELRGB (and the other RGB color selection functions such as **SETPIXELSRGB**, **SETCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (such as **SETPIXELS** and **SETCOLOR**) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPIXELRGB, GETPIXELSRGB, SETCOLORRGB, SETPIXELSRGB

Example

```
! Build as a Graphics ap.
USE DFLIB
INTEGER(2) x, y
INTEGER(4) color
DO i = 10, 30, 10
  SELECT CASE (i)
    CASE(10)
      color = #0000FF
    CASE(20)
      color = #00FF00
    CASE (30)
      color = #FF0000
  END SELECT
! Draw pixels.
  DO y = 50, 180, 2
    status = SETPIXELRGB( x, y, color )
    x      = x + 2
  END DO
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

SETPIXELS

Graphics Subroutine: Sets the color indexes of multiple pixels.

Module: USE DFLIB

Syntax

CALL SETPIXELS (*n, x, y, color*)

n
(Input) INTEGER(4). Number of pixels to set. Sets the number of elements in the other arguments.

x, y
(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to set.

color
(Input) INTEGER(2). Array containing color indexes to set the pixels to.

SETPIXELS sets the pixels specified in the arrays *x* and *y* to the color indexes in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element

refers to the next pixel, and so on.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to **SETPIXELS** with *n* less than 1 are also ignored. **SETPIXELS** is a much faster way to set multiple pixel color indexes than individual calls to **SETPIXEL**.

Unlike **SETPIXELS**, **SETPIXELSRGB** gives access to the full color capacity of the system by using direct color values rather than indexes to a palette. The non-RGB color functions (such as **SETPIXELS** and **SETCOLOR**) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPIXELS, SETPIXEL, SETPIXELSRGB

Example

```
! Build as a Graphics ap.
    USE DFLIB
    INTEGER(2) color(9)
    INTEGER(2) x(9), y(9), i
    DO i = 1, 9
      x(i) = 20 * i
      y(i) = 10 * i
      color(i) = INT2(i)
    END DO
    CALL SETPIXELS(9, x, y, color)
    END
```

SETPIXELSRGB

Graphics Subroutine: Sets multiple pixels to the given Red-Green-Blue (RGB) color.

Module: USE DFLIB

Syntax

CALL SETPIXELSRGB (*n*, *x*, *y*, *color*)

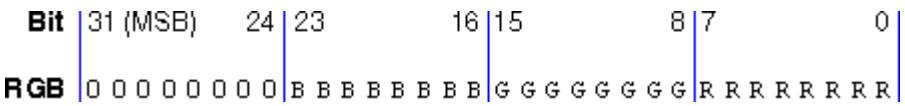
n
(Input) INTEGER(4). Number of pixels to be changed. Determines the number of elements in arrays *x* and *y*.

x, *y*
(Input) INTEGER(2). Parallel arrays containing viewport coordinates of the pixels to set.

color
(Input) INTEGER(4). Array containing the RGB color values to set the pixels to. Range and result depend on the system's display adapter.

SETPIXELSRGB sets the pixels specified in the arrays *x* and *y* to the RGB color values in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three color values, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you set with **SETPIXELSRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

A good use for **SETPIXELSRGB** is as a buffering form of **SETPIXELRGB**, which can improve performance substantially. The example code shows how to do this.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to **SETPIXELSRGB** with *n* less than 1 are also ignored.

SETPIXELSRGB (and the other RGB color selection functions such as **SETPIXELRGB** and **SETCOLORRGB**) sets colors to values chosen from the entire available range. The non-RGB color functions (such as **SETPIXELS** and **SETCOLOR**) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETPIXELSRGB, SETPIXELRGB, GETPIXELRGB, SETPIXELS

Example

```
! Buffering replacement for SetPixelRGB and
! SetPixelRGB_W. This can improve performance by
! doing batches of pixels together.

USE DFLIB
PARAMETER (I$SIZE = 200)
```

```
INTEGER(4) bn, bc(I$SIZE), status
INTEGER(2) bx(I$SIZE),by(I$SIZE)

bn = 0
DO i = 1, I$SIZE
  bn = bn + 1
  bx(bn) = i
  by(bn) = i
  bc(bn) = GETCOLORRGB()
  status = SETCOLORRGB(bc(bn)+1)
END DO
CALL SETPIXELSRGB(bn,bx,by,bc)
END
```

SETTEXTCOLOR

Graphics Function: Sets the current text color index.

Module: USE DFLIB

Syntax

result = **SETTEXTCOLOR** (*index*)

index
(Input) INTEGER(2). Color index to set the text color to.

Results:

The result type is INTEGER(2). The result is the previous text color index.

SETTEXTCOLOR sets the current text color index. The default value is 15, which is associated with white unless the user remaps the palette. **GETTEXTCOLOR** returns the text color index set by **SETTEXTCOLOR**. **SETTEXTCOLOR** affects text output with **OUTTEXT**, **WRITE**, and **PRINT**.

The background color index is set with **SETBKCOLOR** and returned with **GETBKCOLOR**. The color index of graphics over the background color is set with **SETCOLOR** and returned with **GETCOLOR**. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use **SETTEXTCOLORRGB**, **SETBKCOLORRGB**, and **SETCOLORRGB**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETTEXTCOLOR, REMAPPALETTERGB, SETCOLOR, SETTEXTCOLORRGB

Example

```
! Build as a Graphics ap.
USE DFLIB
```

```
INTEGER(2) oldtc
oldtc = SETTEXTCOLOR(INT2(2)) ! green
WRITE(*,*) "hello, world"
END
```

SETTEXTCOLORRGB

Graphics Function: Sets the current text color to the specified Red-Green-Blue (RGB) value.

Module: USE DFLIB

Syntax

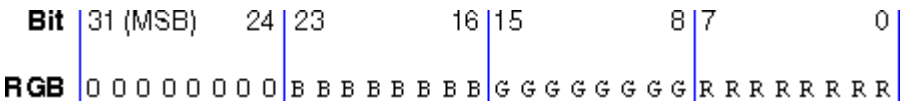
result = **SETTEXTCOLORRGB** (*color*)

color
(Input) INTEGER(4). RGB color value to set the text color to. Range and result depend on the system's display adapter.

Results:

The result type is INTEGER(4). The result is the previous text RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with **SETTEXTCOLORRRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETTEXTCOLORRGB sets the current text RGB color. The default value is #00FFFFFF, which is full-intensity white. **SETTEXTCOLORRGB** sets the color used by **OUTTEXT**, **WRITE**, and **PRINT**. It does not affect the color of text output with the **OUTGTEXT** font routine. Use **SETCOLORRGB** to change the color of font output.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. **SETCOLORRGB** sets the RGB color value of graphics over the background color, used by the graphics functions such as **ARC**, **FLOODFILLRGB**, and **OUTGTEXT**.

SETTEXTCOLORRGB (and the other RGB color selection functions **SETBKCOLORRGB** and **SETCOLORRGB**) sets the color to a value chosen from the entire available range. The non-RGB color functions (**SETTEXTCOLOR**, **SETBKCOLOR**, and **SETCOLOR**) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: SETBKCOLORRGB, SETCOLORRGB, GETTEXTCOLORRGB, GETWINDOWCONFIG, OUTTEXT

Example

```
! Build as a Graphics ap.
USE DFLIB
INTEGER(4) oldtc

oldtc = SETTEXTCOLORRGB(#000000FF)
WRITE(*,*) 'I am red'
oldtc = SETTEXTCOLORRGB(#0000FF00)
CALL OUTTEXT ('I am green'//CHAR(13)//CHAR(10))
oldtc = SETTEXTCOLORRGB(#00FF0000)
PRINT *, 'I am blue'
END
```

SETTEXTPOSITION

Graphics Subroutine: Sets the current text position to a specified position relative to the current text window.

Module: USE DFLIB

Syntax

CALL SETTEXTPOSITION (*row*, *column*, *t*)

row
(Input) INTEGER(2). New text row position.

column
(Input) INTEGER(2). New text column position.

t
(Output) Derived type rccoord. Previous text position. The derived type rccoord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE rccoord
  INTEGER(2) row ! Row coordinate
  INTEGER(2) col ! Column coordinate
END TYPE rccoord
```

Subsequent text output with the **OUTTEXT** function (as well as standard console I/O statements, such as **PRINT** and **WRITE**) begins at the point (*row*, *column*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: CLEARSCREEN, GETTEXTPOSITION, OUTTEXT, SCROLLTEXTWINDOW, SETTEXTWINDOW, WRAPON

Example

```
USE DFLIB
TYPE (rccoord) curpos

WRITE(*,*) "Original text position"
CALL SETTEXTPOSITION (INT2(6), INT2(5), curpos)
WRITE (*,*) 'New text position'
END
```

SETTEXTWINDOW

Graphics Subroutine: Sets the current text window.

Module: USE DFLIB

Syntax

CALL SETTEXTWINDOW (*r1*, *c1*, *r2*, *c2*)

r1, *c1*
(Input) INTEGER(2). Row and column coordinates for upper-left corner of the text window.

r2, *c2*
(Input) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

SETTEXTWINDOW specifies a window in row and column coordinates where text output to the screen using **OUTTEXT**, **WRITE**, or **PRINT** will be displayed. You set the text location within this window with **SETTEXTPOSITION**.

Text is output from the top of the window down. When the window is full, successive lines overwrite the last line.

SETTEXTWINDOW does not affect the output of the graphics text routine **OUTGTEXT**. Use the **SETVIEWPORT** function to control the display area for graphics output.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETTEXTPOSITION, GETTEXTWINDOW, GRSTATUS, OUTTEXT, SCROLLTEXTWINDOW, SETTEXTPOSITION, SETVIEWPORT, WRAPON

Example

```
USE DFLIB
TYPE (rccoord) curpos

CALL SETTEXTWINDOW(INT2(5), INT2(1), INT2(7), &
                   INT2(40))
CALL SETTEXTPOSITION (INT2(5), INT2(5), curpos)
WRITE(*,*) "Only two lines in this text window"
WRITE(*,*) "so this line will be overwritten"
WRITE(*,*) "by this line"
END
```

SETTIM

Run-Time Function: Sets the system time in your programs.

Module: USE DFLIB

Syntax

result = **SETTIM** (*ihr*, *imin*, *isec*, *i100th*)

ihr
(Input) INTEGER(2). Hour (0 - 23).

imin
(Input) INTEGER(2). Minute (0 - 59).

isec
(Input) INTEGER(2). Second (0 - 59).

i100th
(Input) INTEGER(2). Hundredth of a second (0 - 99).

Results:

The result type is LOGICAL(4). The result is .TRUE. if the system time is changed; .FALSE. if no change is made.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETDAT, GETTIM, SETDAT

Example

```
USE DFLIB
LOGICAL(4) success
success = SETTIM(INT2(21),INT2(53+3),&
                 INT2(14*2),INT2(88))
END
```

SETVIEWORG

Graphics Subroutine: Moves the viewport-coordinate origin (0, 0) to the specified physical point.

Module: USE DFLIB

Syntax

CALL SETVIEWORG (*x*, *y*, *t*)

x, *y*
(Input) INTEGER(2). Physical coordinates of new viewport origin.

t
(Output) Derived type xycoord. Physical coordinates of the previous viewport origin. The derived type xycoord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE xycoord
INTEGER(2) xcoord ! x-coordinate
INTEGER(2) ycoord ! y-coordinate
END TYPE xycoord
```

The xycoord type variable *t*, defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory), returns the physical coordinates of the previous viewport origin.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETCURRENTPOSITION, GETPHYSCOORD, GETVIEWCOORD, GETWINDOWCOORD, GRSTATUS, SETCLIPRGN, SETVIEWPORT

Example

```
USE DFLIB
TYPE ( xycoord ) xy

CALL SETVIEWORG(INT2(30), INT2(30), xy)
```

SETVIEWPORT

Graphics Subroutine: Redefines the graphics viewport by defining a clipping region in the same manner as **SETCLIPRGN** and then setting the viewport-coordinate origin to the upper-left corner of

the region.

Module: USE DFLIB

Syntax

CALL SETVIEWPORT (*x1*, *y1*, *x2*, *y2*)

x1, *y1*
(Input) INTEGER(2). Physical coordinates for upper-left corner of viewport.

x2, *y2*
(Input) INTEGER(2). Physical coordinates for lower-right corner of viewport.

The physical coordinates (*x1*, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangular clipping region. Any window transformation done with the **SETWINDOW** function is relative to the viewport, not the entire screen.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETVIEWCOORD, GETPHYSCOORD, GRSTATUS, SETCLIPRGN, SETVIEWORG, SETWINDOW

Example

```
USE DFLIB
INTEGER(2) upx, upy
INTEGER(2) downx, downy

upx = 0
upy = 30
downx= 250
downy = 100
CALL SETVIEWPORT(upx, upy, downx, downy)
```

SETWINDOW

Graphics Function: Defines a window bound by the specified coordinates.

Module: USE DFLIB

Syntax

result = **SETWINDOW** (*finvert*, *wx1*, *wy1*, *wx2*, *wy2*)

finvert
(Input) LOGICAL(2). Direction of increase of the y-axis. If *finv*ert is .TRUE., the y-axis increases from the window bottom to the window top (as Cartesian coordinates). If *finvert* is

.FALSE., the y-axis increases from the window top to the window bottom (as pixel coordinates).

wx1, *wy1*
(Input) REAL(8). Window coordinates for upper-left corner of window.

wx2, *wy2*
(Input) REAL(8). Window coordinates for lower-right corner of window.

Results:

The result type is INTEGER(2). The result is The result is nonzero if successful; otherwise, 0 (for example, if the program that calls **SETWINDOW** is not in a graphics mode).

The **SETWINDOW** function determines the coordinate system used by all window-relative graphics routines. Any graphics routines that end in **_W** (such as **ARC_W**, **RECTANGLE_W**, and **LINETO_W**) use the coordinate sytem set by **SETWINDOW**.

Any window transformation done with the **SETWINDOW** function is relative to the viewport, not the entire screen.

An arc drawn using inverted window coordinates is not an upside-down version of an arc drawn with the same parameters in a noninverted window. The arc is still drawn counterclockwise, but the points that define where the arc begins and ends are inverted.

If *wx1* equals *wx2* or *wy1* equals *wy2*, **SETWINDOW** fails.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETWINDOWCOORD, SETCLIPRGN, SETVIEWORG, SETVIEWPORT, GRSTATUS, ARC_W, LINETO_W, MOVETO_W, PIE_W, POLYGON_W, RECTANGLE_W

Example

```
USE DFLIB
INTEGER(2) status
LOGICAL(2) invert /.TRUE./
REAL(8) upx /0.0/, upy /0.0/
REAL(8) downx /1000.0/, downy /1000.0/
status = SETWINDOW(invert, upx, upy, downx, downy)
```

SETWINDOWCONFIG

QuickWin Function: Sets the properties of a child window.

Module: USE DFLIB

Syntax

result = **SETWINDOWCONFIG** (*wc*)

wc
(Input) Derived type windowconfig. Contains window properties. The windowconfig derived type is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:

```
TYPE windowconfig
  INTEGER(2) numxpixels   ! Number of pixels on x-axis
  INTEGER(2) numypixels   ! Number of pixels on y-axis
  INTEGER(2) numtextcols  ! Number of text columns
                          ! available
  INTEGER(2) numtextrows  ! Number of text rows
                          ! available
  INTEGER(2) numcolors    ! Number of color indexes
  INTEGER(4) fontsize     ! Size of default font. Set
                            ! to QWIN$EXTENDFONT when using
                            ! using multibyte characters, in
                            ! which case extendfontsize sets
                            ! the sets the font size.
  CHARACTER(80) title ! window title, a C string
! The next three parameters support multibyte character
! sets (such as Japanese)
  CHARACTER(32) extendfontname ! any non-proportionally
                ! spaced font available on the system
  INTEGER(4) extendfontsize ! takes same values as
                ! fontsize, but used for multibyte
                ! character sets when fontsize is set
                ! to QWIN$EXTENDFONT
  INTEGER(4) extendfontattributes   ! font attributes
                       ! such as bold and italic for
                       ! multibyte character sets
END TYPE windowconfig
```

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

If you use **SETWINDOWCONFIG** to set the variables in windowconfig to - 1, the function sets the highest resolution possible for your system, given the other fields you specify, if any. You can set the actual size of the window by specifying parameters that influence the window size: the number of x and y pixels, the number of rows and columns, and the font size. If you do not call **SETWINDOWCONFIG**, the window defaults to the best possible resolution and a font size of 8x16. The number of colors available depends on the video driver used.

If you use **SETWINDOWCONFIG**, you should specify a value for each field (-1 or your own value for the numeric fields and a C string for the title, for example, "words of text"C). Using **SETWINDOWCONFIG** with only some fields specified can result in useless values for the unspecified fields.

If you request a configuration that cannot be set, **SETWINDOWCONFIG** returns .FALSE. and calculates parameter values that will work and are as close as possible to the requested configuration. A second call to **SETWINDOWCONFIG** establishes the adjusted values; for example:

```
status = SETWINDOWCONFIG(wc)
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

If you specify values for all four of the size parameters, *numxpixels*, *numypixel*, *numtextcols*, and *numtextrows*, the font size is calculated by dividing these values.The default font is Courier New and the default font size is 8x16. There is no restriction on font size, except that the window must be large enough to hold it.

Under Standard Graphics, if the resolution specified matches the resolution of the graphics driver (or if -1 is specified for the four size variables), the application starts in Full Screen mode with no window decoration (window decoration includes scroll bars, menu bar, title bar, and message bar) so that the maximum resolution can be fully used. Otherwise, the application starts in a window. You can use ALT+ENTER at any time to toggle between the two modes.

Note that if you are in Full Screen mode and the resolution of the window does not match the resolution of the video driver, graphics output will be slow compared to drawing in a window.

You must call **DISPLAYCURSOR**($GCURSORON) to make the cursor visible after calling **SETWINDOWCONFIG**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, GETWINDOWCONFIG.

Example

```
USE DFLIB
TYPE (windowconfig) wc
LOGICAL status /.FALSE./
! Set the x & y pixels to 800X600 and font size to 8x12
wc.numxpixels  = 800
wc.numypixels  = 600
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors   = -1
wc.title= "This is a test"C
wc.fontsize = #0008000C
status = SETWINDOWCONFIG(wc)  ! attempt to set configuration with above values
   ! if attempt fails, set with system estimated values
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

SETWINDOWMENUQQ

QuickWin Function: Sets a top-level menu as the menu to which a list of current child window names is appended.

Module: USE DFLIB

Syntax

result = **SETWINDOWMENUQQ** (*menuID*)

menuID
(Input) INTEGER(4). Identifies the menu to hold the child window names, starting with 1 as the leftmost menu.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The list of current child window names can appear in only one menu at a time. If the list of windows is currently in a menu, it is removed from that menu. By default, the list of child windows appears at the end of the Window menu.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: APPENDMENUQQ, Using QuickWin, Customizing QuickWin Applications

Example

```
USE DFLIB
TYPE (windowconfig) wc
LOGICAL(4) result, status /.FALSE./
! Set title for child window
wc.numxpixels  = -1
wc.numypixels  = -1
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors   = -1
wc.fontsize    = -1
wc.title= "I am child window name"C
if (.NOT.status) status = SETWINDOWCONFIG(wc)

! put child window list under menu 3 (View)
result = SETWINDOWMENUQQ(3)
END
```

SETWRITEMODE

Graphics Function: Sets the current logical write mode, which is used when drawing lines with the **LINETO**, **POLYGON**, and **RECTANGLE** functions.

Module: USE DFLIB

Syntax

result = **SETWRITEMODE** (*wmode*)

wmode

(Input) INTEGER(2). Write mode to be set. One of the following symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory):

- **$GPSET**: Causes lines to be drawn in the current graphics color. (Default)
- **$GAND**: Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- **$GOR**: Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- **$GPRESET**: Causes lines to be drawn in the color that is the logical NOT of the current graphics color.
- **$GXOR** : Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.
- In addition, one of the following binary raster operation constants can be used (described in the online documentation for the WIN32 API SetROP2):
  - **$GR2_BLACK**
  - **$GR2_NOTMERGEPEN**
  - **$GR2_MASKNOTPEN**
  - **$GR2_NOTCOPYPEN** (same as **$GPRESET**)
  - **$GR2_MASKPENNOT**
  - **$GR2_NOT**
  - **$GR2_XORPEN** (same as **$GXOR**)
  - **$GR2_NOTMASKPEN**
  - **$GR2_MASKPEN** (same as **$GAND**)
  - **$GR2_NOTXORPEN**
  - **$GR2_NOP**
  - **$GR2_MERGENOTPEN**
  - **$GR2_COPYPEN** (same as **$GPSET**)
  - **$GR2_MERGEPENNOT**
  - **$GR2_MERGEPEN** (same as **$GOR**)
  - **$GR2_WHITE**

Results:

The result type is INTEGER(2). The result is the previous write mode if successful; otherwise, -1.

The current graphics color is set with **SETCOLORRGB** (or **SETCOLOR**) and the current background color is set with **SETBKCOLORRGB** (or **SETBKCOLOR**). As an example, suppose you set the background color to yellow (#00FFFF) and the graphics color to purple (#FF00FF) with the following commands:

```
oldcolor = SETBKCOLORRGB(#00FFFF)
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB(#FF00FF)
```

If you then set the write mode with the $GAND option, lines are drawn in red (#0000FF); with the $GOR option, lines are drawn in white (#FFFFFF); with the $GXOR option, lines are drawn in turquoise (#FFFF00); and with the $GPRESET option, lines are drawn in green (#00FF00). Setting the write mode to $GPSET causes lines to be drawn in the graphics color.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETWRITEMODE, GRSTATUS, LINETO, POLYGON, PUTIMAGE, RECTANGLE, SETCOLOR, SETLINESTYLE

Example

```
! Build as a Graphics ap.
USE DFLIB
INTEGER(2) result, oldmode
INTEGER(4) oldcolor
TYPE (xycoord) xy

oldcolor = SETBKCOLORRGB(#00FFFF)
CALL CLEARSCREEN ($GCLEARSCREEN)
oldcolor = SETCOLORRGB(#FF00FF)
CALL MOVETO(INT2(0), INT2(0), xy)
result = LINETO(INT2(200), INT2(200)) ! purple

oldmode = SETWRITEMODE( $GAND)
CALL MOVETO(INT2(50), INT2(0), xy)
result = LINETO(INT2(250), INT2(200)) ! red
END
```

SETWSIZEQQ

QuickWin Function: Sets the size and position of a window.

Module: USE DFLIB

Syntax

result = **SETWSIZEQQ** (*unit*, *winfo*)

unit
(Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5, and 6 refer to the default startup window only if the program does not explicitly open them with the **OPEN** statement. To set the size of the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory).

winfo
(Input) Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type qwinfo is defined in DFLIB.F90 as follows:

```
TYPE QWINFO
   INTEGER(2) TYPE  ! request type
   INTEGER(2) X     ! x coordinate for upper left
   INTEGER(2) Y     ! y coordinate for upper left
```

```
    INTEGER(2) H     ! window height
    INTEGER(2) W     ! window width
END TYPE QWINFO
```

This function's behavior depends on the value of `qwinfo.type`, which can be any of the following:

- **QWIN$MIN**: Minimizes the window.
- **QWIN$MAX**: Maximizes the window.
- **QWIN$RESTORE**: Restores the minimized window to its previous size.
- **QWIN$SET**: Sets the window's position and size according to the other values in qwinfo.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width specified for a frame window reflects the actual size in pixels of the frame window *including* any borders, menus, and status bar at the bottom.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, GETWSIZEQQ.

Example

```
USE DFLIB
LOGICAL(4) result
INTEGER(2) numfonts, fontnum
TYPE (qwinfo) winfo
TYPE (xycoord) pos
! Maximize frame window
winfo.TYPE = QWIN$MAX
result = SETWSIZEQQ(QWIN$FRAMEWINDOW, winfo)
! Maximize child window
result=SETWSIZEQQ(0, winfo)
numfonts = INITIALIZEFONTS( )
fontnum = SETFONT ('t''Arial''h50w34i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT("BIG Window")
END
```

SHAPE

Inquiry Intrinsic Function (Generic): Returns the shape of an array or scalar argument.

Syntax

result = **SHAPE** (*source*)

source
(Input) 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.

Results:

The result is a rank-one default integer array whose size is equal to the rank of *source*. The value of the result is the shape of *source*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SIZE

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).

The following shows another example:

```
INTEGER VEC(2)
REAL array(3:10, -1:3)
VEC = SHAPE(array)
WRITE(*,*) VEC ! prints   8       5
END
!
! Check if a mask is conformal with an array
REAL, ALLOCATABLE :: A(:,:,:)
LOGICAL, ALLOCATABLE :: MASK(:,:,:)
INTEGER B(3), C(3)
LOGICAL conform
ALLOCATE (A(5, 4, 3))
ALLOCATE (MASK(3, 4, 5))
! Check if MASK and A allocated. If they are, check
! that they have the same shape (conform).
IF(ALLOCATED(A) .AND. ALLOCATED(MASK)) THEN
  B = SHAPE(A); C = SHAPE(MASK)
  IF ((B(1) .EQ. C(1)) .AND. (B(2) .EQ. C(2))       &
       .AND. (B(3) .EQ. C(3))) THEN
     conform = .TRUE.
  ELSE
     conform = .FALSE.
  END IF
END IF
WRITE(*,*) conform  ! prints F
END
```

SHORT

Portability Function: Converts an INTEGER(4) value into an equivalent INTEGER(2) type.

Module:USE DFPORT

Syntax

result = **SHORT** (*int4*)

int4
(Input) INTEGER(4). Value to be converted.

Results:

The result type is INTEGER(2). The result is equal to the lower 16 bits of *int4*. If the *int4* value is greater than 32,767, the converted INTEGER(2) value is not equal to the original.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INT, TYPE, Portability Library

Example

```
      USE DFPORT
      INTEGER(4) this_one
      INTEGER(2) that_one
      READ(*,*) this_one
      THAT_ONE = SHORT(THIS_ONE)
      WRITE(*,10) THIS_ONE, THAT_ONE
 10   FORMAT (X," Long integer: ", I16, " Short integer: ", I16)
      END
```

SIGN

Elemental Intrinsic Function (Generic): Returns the absolute value of the first argument times the sign of the second argument.

Syntax

result = **SIGN** (*a*, *b*)

a
(Input) 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 /assume:minus0 compiler option is specified and the processor can distinguish between positive and negative real zero, 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 [2]	INTEGER(8)	INTEGER(8)
SIGN	REAL(4)	REAL(4)
DSIGN	REAL(8)	REAL(8)
QSIGN [3]	REAL(16)	REAL(16)

[1] Or JISIGN. For compatibility with older versions of Fortran, ISIGN can also be specified as a generic function.
[2] Alpha only
[3] VMS and U*X

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ABS

Examples

SIGN (4.0, -6.0) has the value -4.0.

SIGN (-5.0, 2.0) has the value 5.0.

The following shows another example:

```
c = SIGN (5.2, -3.1)   ! returns -5.2
c = SIGN (-5.2, -3.1)  ! returns -5.2
c = SIGN (-5.2, 3.1)   ! returns  5.2
```

SIN

Elemental Intrinsic Function (Generic): Produces a sine (with the result in radians).

Syntax

result = **SIN** (*x*)

x
(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi. (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 [1]	REAL(16)	REAL(16)
CSIN [2]	COMPLEX(4)	COMPLEX(4)
CDSIN [3]	COMPLEX(8)	COMPLEX(8)

[1] VMS and U*X
[2] The setting of compiler option /real_size can affect CSIN.
[3] 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.

SIND

Elemental Intrinsic Function (Generic): Produces a sine (with the result in degrees).

Syntax

result = **SIND** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

SIND (2.0) has the value 3.4899496E-02.

SIND (0.8) has the value 1.3962180E-02.

SINH

Elemental Intrinsic Function (Generic): Produces a hyperbolic sine.

Syntax

result = **SINH** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

SINH (2.0) has the value 3.626860.

SINH (0.8) has the value 0.8881060.

SIGNAL

Portability Function: Controls interrupt signal handling. Changes the action for a specified signal.

Module: USE DFPORT

Syntax

result = **SIGNAL** (*signum*, *proc*, *flag*)

signum
(Input) INTEGER(4). Number of the signal to change. The numbers and symbolic names are listed in a table below.

proc
(Input) Name of an external signal-processing routine. This routine is called only if *flag* is negative.

flag
(Input) INTEGER(4). If negative, the user's *proc* routine is called. If 0, the signal retains its default action; if 1, the signal should be ignored.

Results:

The result type is INTEGER(4). The result is the previous value of *proc* associated with the specified signal. For example, if the previous value of *proc* was SIG_IGN, the return value is also SIG_IGN. You can use this return value in subsequent calls to **SIGNAL** if the signal number supplied is invalid, if the flag value is greater than 1, or to restore a previous action definition.

A return value of SIG_ERR indicates an error, in which case a call to **IERRNO** returns EINVAL. If the signal number supplied is invalid, or if the flag value is greater than 1, **SIGNAL** returns -(EINVAL) and a call to **IERRNO** returns EINVAL.

An initial signal handler is in place at startup for SIGFPE (signal 8); its address is returned the first time **SIGNAL** is called for SIGFPE. No other signals have initial signal handlers.

Be careful when you use **SIGNALQQ** or the C signal function to set a handler, and then use the Portability **SIGNAL** function to retrieve its value. If **SIGNAL** returns an address that was not previously set by a call to **SIGNAL**, you cannot use that address with either **SIGNALQQ** or C's signal function, nor can you call it directly. You can, however, use the return value from **SIGNAL** in a subsequent call to **SIGNAL**. This allows you to restore a signal handler, no matter how the original signal handler was set.

All signal handlers are called with a single integer argument, that of the signal number actually received. Usually, when a process receives a signal, it terminates. With the **SIGNAL** function, a user procedure is called instead. The signal handler routine must accept the signal number integer argument, even if it does not use it. If the routine does not accept the signal number argument, the stack will not be properly restored after the signal handler has executed.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. There are certain restrictions as to which functions you can use in your signal-handler routine:

- Do not do either low-level (such as **FGETC**) or high-level (such as **READ**) I/O.
- Do not call heap routines or any routine that uses the heap routines (such as **MALLOC**, **ALLOCATE**).
- Do not use any function that generates a system call (such as **TIME**).

SIGKILL can be neither caught nor ignored.

The following table lists signals, their names and values:

Symbolic name	Number	Description
SIGABRT	6	Abnormal termination
SIGFPE	8	Floating-point error
SIGKILL	9	Kill process
SIGILL	4	Illegal instruction
SIGINT	2	CTRL+C signal
SIGSEGV	11	Illegal storage access
SIGTERM	15	Termination request

The default action for all signals is to terminate the program with exit code

ABORT does not assert the SIGABRT signal. The only way to assert SIGABRT or SIGTERM is to use **KILL**.

SIGNAL can be used to catch SIGFPE exceptions, but it cannot be used to access the error code that caused the SIGFPE. To do this, use **SIGNALQQ** instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SIGNALQQ

Example

```
    USE dfport
    EXTERNAL h_abort
    INTEGER(4) iret1, iret2, procnum
    iret1 = SIGNAL(SIGABRT, h_abort, -1)
    WRITE(*,*) 'Set signal handler. Return = ', iret1

    iret2 = KILL(procnum, SIGABRT)
    WRITE(*,*) 'Raised signal. Return = ', iret2
    END
!
!  Signal handler routine
!
    INTEGER(4) FUNCTION h_abort (sig_num)
    INTEGER(4) sig_num

    WRITE(*,*) 'In signal handler for SIG$ABORT'
    WRITE(*,*) 'signum = ', sig_num
    h_abort = 1
    END
```

SIGNALQQ

Run-Time Function: Registers the function to be called if an interrupt signal occurs.

Module: USE DFLIB

Syntax

result = **SIGNALQQ** (*sig*, *func*)

sig
(Input) INTEGER(2). Interrupt type. One of the following constants, defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory):

- **SIG$ABORT**: Abnormal termination
- **SIG$FPE**: Floating-point error
- **SIG$ILL**: Illegal instruction
- **SIG$INT**: CTRL+C SIGNAL
- **SIG$SEGV**: Illegal storage access
- **SIG$TERM**: Termination request

func
(Input) Character*(*). Name of function to be executed on interrupt.

Results:

The result type is INTEGER(4). The result is a positive integer if successful; otherwise, -1 (SIG$ERR).

SIGNALQQ installs the function *func* as the handler for a signal of the type specified by *sig*. If you do not install a handler, the system by default terminates the program with exit code 3 when an interrupt signal occurs.

The argument *func* is the name of a function and must be declared with either the **EXTERNAL** or **IMPLICIT** statements, or have an explicit interface. A function described in an **INTERFACE** block is **EXTERNAL** by default, and does not need to be declared **EXTERNAL**.

Note: All signal-handler functions must be declared with the cDEC$ ATTRIBUTES C option.

When an interrupt occurs, except a **SIG$FPE** interrupt, the *sig* argument **SIG$INT** is passed to *func*, and then *func* is executed.

When a SIG$FPE interrupt occurs, the function *func* is passed two arguments: SIG$FPE and the floating-point error code (for example, FPE$ZERODIVIDE or FPE$OVERFLOW) which identifies the type of floating-point exception that occurred. The floating-point error codes begin with the prefix FPE$ and are defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory. Floating-point exceptions are described and discussed in The Floating-Point Environment in the *Programmer's Guide*.

If *func* returns, the calling process resumes execution immediately after the point at which it received the interrupt signal. This is true regardless of the type of interrupt or operating mode.

Because signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. Therefore, do not call heap routines or any routine that uses the heap routines (for example, I/O routines, **ALLOCATE**, and **DEALLOCATE**).

To test your signal handler routine you can generate interrupt signals by calling **RAISEQQ**, which causes your program either to branch to the signal handlers set with **SIGNALQQ**, or to perform the system default behavior if **SIGNALQQ** has set no signal handler.

The example below demonstrates a signal handler for SIG$ABORT. A sample signal handler for SIG$FPE is given in Handling Floating-Point Exceptions in the *Programmer's Guide*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RAISEQQ, SIGNAL, KILL

Example

```
!  This program shows a signal handler for
!  SIG$ABORT
USE DFLIB
```

```
INTERFACE
  FUNCTION h_abort (signum)
    !DEC$ATTRIBUTES C :: h_abort
    INTEGER(4) h_abort
    INTEGER(2) signum
  END FUNCTION
END INTERFACE

INTEGER(2) i2ret
INTEGER(4) i4ret

i4ret = SIGNALQQ(SIG$ABORT, h_abort)
WRITE(*,*) 'Set signal handler. Return = ', i4ret

i2ret = RAISEQQ(SIG$ABORT)
WRITE(*,*) 'Raised signal. Return = ', i2ret
END
!
!      Signal handler routine
!
INTEGER(4) FUNCTION h_abort (signum)
  !DEC$ATTRIBUTES C :: h_abort
  INTEGER(2) signum
  WRITE(*,*) 'In signal handler for SIG$ABORT'
  WRITE(*,*) 'signum = ', signum
  h_abort = 1
END
```

SIZE

Inquiry Intrinsic Function (Generic): Returns the total number of elements in an array, or the extent of an array along a specified dimension.

Syntax

result = **SIZE** (*array* [, *dim*])

array
(Input) 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
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.

Results:

The result is a scalar of type integer. 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*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SHAPE

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.

The following shows another example:

```
REAL(8) array (3:10, -1:3)
INTEGER i
i = SIZE(array, DIM = 2) ! returns 5
i = SIZE(array)          ! returns 40
```

SIZEOF

Inquiry Intrinsic Function (Specific): Returns the number of bytes of storage used by the argument. This is a specific function with no generic name.

Syntax

result = **SIZEOF** (*x*)

x
Can be a scalar or array (of any data type). It must *not* be an assumed-size array.

Results:

The result type is INTEGER(4) on Intel processors; INTEGER(8) on Alpha processors. 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
```

SLEEP

Portability Subroutine: Suspends the execution of a process for a specified interval.

Module: USE DFPORT

Syntax

CALL SLEEP (*time*)

time
(Input) INTEGER(4). Length of time, in seconds, to suspend the calling process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SLEEPQQ

Example

```
USE DFPORT
integer(4) hold_time
hold_time = 1 !lets the loop execute
DO WHILE (hold_time .NE. 0)
  write(*,'(A)') "Enter the number of seconds to suspend"
  read(*,*) hold_time
  CALL SLEEP (hold_time)
END DO
END
```

SLEEPQQ

Run-Time Subroutine: Delays execution of the program for a specified duration.

Module: USE DFLIB

Syntax

CALL SLEEPQQ (*duration*)

duration
(Input) INTEGER(4). Number of milliseconds the program is to sleep (delay program execution).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE DFLIB
INTEGER(4) delay, freq, duration
delay    = 2000
freq     = 4000
duration = 1000
CALL SLEEPQQ(delay)
CALL BEEPQQ(freq, duration)
END
```

SNGL

See REAL.

SORTQQ

Run-Time Subroutine: Sorts a one-dimensional array. The array elements cannot be derived types or record structures.

Module: USE DFLIB

Syntax

CALL SORTQQ (*adrarray*, *count*, *size*)

adrarray
(Input) INTEGER(4). Address of the array (returned by **LOC**).

count
(Input; output) INTEGER(4). On input, number of elements in the array to be sorted. On output, number of elements actually sorted.

size
(Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory), specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$REAL4	REAL(4) or equivalent
SRT$REAL8	REAL(8) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

To be certain that **SORTQQ** is successful, compare the value returned in *count* to the value you provided. If they are the same, then **SORTQQ** sorted the correct number of elements.

Caution: The location of the array must be passed by address using the **LOC** function. This defeats Fortran type-checking, so you must make certain that the *count* and *size* arguments are correct.

If you pass invalid arguments, **SORTQQ** attempts to sort random parts of memory. If the

memory it attempts to sort is allocated to the current process, that memory is sorted; otherwise, the operating system intervenes, the program is halted, and you get a General Protection Violation message.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BSEARCHQQ, LOC

Example

```
!    Sort a 1-D array
!
USE DFLIB
INTEGER(2) array(10)
INTEGER(2) i
DATA ARRAY /143, 99, 612, 61, 712, 9112, 6, 555, 2223, 67/
!    Sort the array
Call SORTQQ (LOC(array), 10, SRT$INTEGER2)
!    Display the sorted array
DO i = 1, 10
  WRITE (*, 9000) i, array (i)
9000 FORMAT(1X, ' Array(',I2, '): ', I5)
END DO
END
```

SPACING

Elemental Intrinsic Function (Generic): Returns the absolute spacing of model numbers near the argument value.

Syntax

result = **SPACING** (*x*)

x
(Input) 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 Model for Real Data. If the result value is outside of the real model range, the result is **TINY**(*x*).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: TINY, RRSPACING, Data Representation Models

Examples

If 3.0 is a REAL(4) value, SPACING (3.0) has the value 2^{-22}.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = SPACING(3.0)   ! returns 2.384186E-07
res4 = SPACING(-3.0)  ! returns 2.384186E-07
r2   = 487923.3
res8 = SPACING(r2)    ! returns 5.820766091346741E-011
```

SPLITPATHQQ

Run-Time Function: Breaks a file path or directory path into its components.

Module: USE DFLIB

Syntax

result = **SPLITPATHQQ** (*path*, *drive*, *dir*, *name*, *ext*)

path
(Input) Character*(*). Path to be broken into components. Forward slashes (/), backslashes (\), or both can be present in *path*.

drive
(Output) Character*(*). Drive letter followed by a colon.

dir
(Output) Character*(*). Path of directories, including the trailing slash.

name
(Output) Character*(*). Name of file or, if no file is specified in *path*, name of the lowest directory. If a filename, does not include an extension.

ext
(Output) Character*(*). Filename extension, if any, including the leading period (.).

Results:

The result type is INTEGER(4). The result is the length of *dir*.

The *path* parameter can be a complete or partial file specification.

$MAXPATH is a symbolic constant defined in module DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as 260.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FULLPATHQQ

Example

```
USE DFLIB
CHARACTER($MAXPATH) buf
CHARACTER(3)        drive
CHARACTER(256)      dir
CHARACTER(256)      name
CHARACTER(256)      ext
CHARACTER(256)      file

INTEGER(4)          length

buf = 'b:\fortran\test\runtime\tsplit.for'
length = SPLITPATHQQ(buf, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
file = 'partial.f90'
length = SPLITPATHQQ(file, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext

END
```

SPREAD

Transformational Intrinsic Function (Generic): Creates a replicated array with an added dimension by making copies of existing elements along a specified dimension.

Syntax

result = **SPREAD** (*source*, *dim*, *ncopies*)

source
(Input) Must be a scalar or array (of any data type). The rank must be less than 7.

dim
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PACK, RESHAPE

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

```
  [ 3  4  5 ]
  [ 3  4  5 ]
  [ 3  4  5 ]
  [ 3  4  5 ].
```

SPREAD (B, DIM=2, NCOPIES=NC) produces the array

```
[3  3  3  3 ]
[4  4  4  4 ]
[5  5  5  5 ].
```

The following shows another example:

```
 INTEGER AR1(2, 3), AR2(3, 2)
 AR1 = SPREAD((/1,2,3/),DIM= 1,NCOPIES= 2) ! returns
                                           ! 1 2 3
                                           ! 1 2 3
 AR2 = SPREAD((/1,2,3/), 2, 2) ! returns   1 1
                               !           2 2
                               !           3 3
```

SQRT

Elemental Intrinsic Function (Generic): Derives the square root of its argument.

Syntax

result = **SQRT** (*x*)

x

(Input) 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 [1]	REAL(16)	REAL(16)
CSQRT [2]	COMPLEX(4)	COMPLEX(4)
CDSQRT [3]	COMPLEX(8)	COMPLEX(8)

[1] VMS and U*X
[2] The setting of compiler option /real_size can affect CSQRT.
[3] This function can also be specified as ZSQRT.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

SQRT (16.0) has the value 4.0.

SQRT (3.0) has the value 1.732051.

The following shows another example:

```
! Calculate the hypotenuse of a right triangle
! from the lengths of the other two sides.
REAL sidea, sideb, hyp
sidea = 3.0
sideb = 4.0
hyp = SQRT (sidea**2 + sideb**2)
WRITE (*, 100) hyp
100 FORMAT (/ ' The hypotenuse is ', F10.3)
END
```

SRAND

Portability Subroutine: Seeds the random number generator used with **IRAND** and **RAND**.

Module: USE DFPORT

Syntax

CALL SRAND (*iseed*)
CALL SRAND (*rseed*)

iseed
(Input) INTEGER(4). Any value.

rseed
Input) REAL(4). Any value.

SRAND seeds the random number generator used with **IRAND** and **RAND**. Calling **SRAND** is equivalent to calling **IRAND** or **RAND** with a new seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RAND, IRAND, RANDOM_NUMBER, RANDOM_SEED

Example

```
!  How many random numbers out of 100 will be between .5 and .6?
   USE DFPORT
   ICOUNT = 0
   CALL SRAND(123.4567)
   DO I = 1, 100
     X = RAND(0.0)
     IF ((X>.5).AND.(x<.6)) ICOUNT = ICOUNT + 1
   END DO
   WRITE(*,*) ICOUNT, "numbers between .5 and .6!"
```

SSWRQQ (x86 only)

Run-Time Subroutine: Returns the floating-point processor status word. This routine is only available on Intel® processors.

Module: USE DFLIB

Syntax

CALL SSWRQQ (*status*)

status
(Output) INTEGER(2). Floating-point processor status word.

SSWRQQ performs the same function as the run-time subroutine **GETSTATUSFPQQ** and is

provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LCWRQQ, GETSTATUSFPQQ

Example

```
USE DFLIB
INTEGER(2) status
CALL SSWRQQ (status)
```

STAT

Portability Function: Returns detailed information about a file.

Module: USE DFPORT

Syntax

result = **STAT** (*name*, *statb*)

name
(Input) Character*(*). Name of the file to examine.

statb
(Output) INTEGER(4). One-dimensional array with a size of 12.

Results:

The result type is INTEGER(4). The result is zero if the inquiry was successful; otherwise, the error code ENOENT (the specified file could not be found). For a list of other error codes, see IERRNO.

The elements of *statb* contain the following values:

Element	Description	Notes
statb(1)	Device file resides on	Always 0
statb(2)	File Inode number	Always 0
statb(3)	Access mode of the file	(See following table)
statb(4)	Number of hard links	Always 1
statb(5)	User ID of owner	Always 1
statb(6)	Group ID of owner	Always 1

statb(7)	Raw device file resides on	Always 0
statb(8)	Size of the file in bytes	
statb(9)	Time when the file was last accessed	(Only available on non-FAT file systems; undefined on FAT systems)
statb(10)	Time when the file was last modified	
statb(11)	Time of last file status change	Same as stat(10)
statb(12)	Blocksize	Always 1

Times are in the same format returned by the **TIME** function (number of seconds since 00:00:00 Greenwich mean time, January 1, 1970).

Access mode (the third element of *statb*) is a bitmap consisting of an IOR of the following constants:

Symbolic name	**Constant**	**Description**	**Notes**
S_IFMT	O'0170000'	Type of File	
S_IFDIR	O'0040000'	Directory	
S_IFCHR	O'0020000'	Character Special	Never set
S_IFBLK	O'0060000'	Block Special	Never set
S_IFREG	O'0100000'	Regular	
S_IFLNK	O'0120000'	Symbolic Link	Never set
S_IFSOCK	O'0140000'	Socket	Never set
S_ISUID	O'0004000'	Set User ID on Execution	Never set
S_ISGID	O'0002000'	Set Group ID on Execution	Never set
S_ISVTX	O'0001000'	Save Swapped Text	Never set
S_IRWXU	O'0000700'	Owner's File Permissions	
S_IRUSR,	O'0000400'	Owner Read	Always true

S_IREAD		Permission	
S_IWUSR, S_IWRITE	O'0000200'	Owner Write Permission	
S_IXUSR, S_IEXEC	O'0000100'	Owner Execute Permission	Based on file extension (.EXE, .COM, .CMD, or .BAT)
S_IRWXG	O'0000070'	Group's File Permissions	Same as S_IRWXU
S_IRGRP	O'0000040'	Group Read Permission	Same as S_IRUSR
S_IWGRP	O'0000020'	Group Write Permission	Same as S_IWUSR
S_IXGRP	O'0000010'	Group Execute Permission	Same as S_IXUSR
S_IRWXO	O'0000007'	Other's File Permissions	Same as S_IRWXU
S_IROTH	O'0000004'	Other Read Permission	Same as S_IRUSR
S_IWOTH	O'0000002'	Other Write Permission	Same as S_IWUSR
S_IXOTH	O'0000001'	Other Execute Permission	Same as S_IXUSR

STAT returns the same information as **FSTAT**, but accesses files by name instead of external unit number.

Note: The **INQUIRE** statement also provides information about file properties.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: INQUIRE, GETFILEINFOQQ

Example

```
USE DFPORT
CHARACTER*12 file_name
INTEGER(4) info_array(12)
print *, 'Enter file to examine: '
read *, file_name
ISTATUS = STAT (file_name, info_array)
if (.not. istatus) then
  print *, info_array
```

```
else
  print *, 'Error = ',istatus
end if
end
```

Statement Function

Statement: Defines a function in a single statement in the same program unit in which the procedure is referenced.

Syntax

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FUNCTION, Argument Association, Use and Host Association

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.
```

The following shows another example:

```
Add (a, b) = a + b
REAL(4) y, x(6)
. . .
DO n = 2, 6
  x(n) = Add (y, x(n-1))
END DO
```

STATIC

Statement and Attribute: Controls the storage allocation of variables in subprograms (as does AUTOMATIC). Variables declared as STATIC and allocated in memory reside in the static storage area, rather than in the stack storage area. Equivalent to the Fortran 90 **SAVE** attribute and the C **static** attribute.

The STATIC attribute can be specified in a type declaration statement or a **STATIC** statement, and takes one of the following forms:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **STATIC** [*att-ls*,] :: *v* [, *v*]...

Statement:

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

STATIC declarations only affect how data is allocated in storage.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

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 following ways:

- As a keyword in a **FUNCTION** or **SUBROUTINE** statement
- As a compiler option
- As an option in an **OPTIONS** statement

To override any compiler option that may affect variables, explicitly specify the variables as 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 STATIC more than once in the same scoping unit.

If the variable is a pointer, STATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as STATIC. The following table shows these restrictions:

Variable	**STATIC**
Dummy argument	No
Automatic object	No
Common block item	Yes
Use-associated item	No
Function result	No
Component of a derived type	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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: AUTOMATIC, SAVE, Type declaration statements, Compatible attributes, RECURSIVE, /recursive, OPTIONS, POINTER, Modules and Module Procedures

Examples

The following example shows a type declaration statement specifying the STATIC attribute:

```
INTEGER, STATIC :: ARRAY_A
```

The following example uses a **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
...

 INTEGER N1, N2
 N1 = -1
 DO WHILE (N1)
   N2 = N1*2
   call sub1(N1, N2)
   read *, N1
 END DO
 CONTAINS
 SUBROUTINE sub1 (iold, inew)
 INTEGER, intent(INOUT):: iold
 integer, STATIC ::N3
 integer, intent(IN) :: inew
 if (iold .eq. -1) then
   N3 = iold
 end if
 print *, 'New: ', inew, 'N3: ',N3
 END subroutine
 !
 END
```

STOP

Statement: Terminates program execution before the end of the program unit.

Syntax

STOP [*stop-code*]

stop-code
(Optional) A 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.)

Effect on Windows NT and Windows 95 Systems

If you specify *stop-code*, the effect differs depending on its form, as follows:

- If *stop-code* is specified as a character constant, the **STOP** statement writes the specified message to the standard error device and terminates program execution. The program returns a status of zero to the operating system.

- If *stop-code* is specified as a string of digits, the **STOP** statement writes the following to the standard error device and terminates program execution:

  ```
  Return code stop-code
  ```

  In QuickWin programs, the following is displayed in a message box:

  ```
  Program terminated with Exit Code stop-code
  ```

  In both cases, the program returns a status of *stop-code* to the operating system as an integer.

If you do not specify *stop-code*, the **STOP** statement writes the following default message to the standard error device and terminates program execution:

```
Stop - Program terminated.
```

The program returns a status of zero to the operating system.

Effect on OpenVMS Systems

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.

Effect on DIGITAL UNIX Systems

If you specify *stop-code*, the **STOP** statement writes the specified message to the standard error device and terminates program execution. The program returns a status of zero to the operating system.

If you do not specify *stop-code*, no message is output.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: EXIT

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
```

The following shows another example:

```
      OPEN(1,FILE='file1.dat', status='OLD', ERR=100)
      . . .
 100  STOP 'ERROR DETECTED!'
      END
```

STRICT and NOSTRICT

Compiler Directive: STRICT disables language features not found in the Fortran 90 language standard. **NOSTRICT** (the default) enables these features.

Syntax

***c*DEC$ STRICT**
***c*DEC$ NOSTRICT**

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

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. **STRICT** and **NOSTRICT** 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.

The following forms are also allowed: `!MS$STRICT` and `!MS$NOSTRICT`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: General Compiler Directives

Example

```
! 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
```

STRUCTURE...END STRUCTURE

Statement: Defines the field names, types of data within fields, and order and alignment of fields within a record structure. Fields and structures can be initialized, but records cannot be initialized.

Syntax

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

  These are ordinary Fortran data type declarations.

- Substructure declarations

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.

- Union declarations

  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

The Fortran 90 derived type replaces **STRUCTURE** and **RECORD** constructs, and should be used in writing new code. See Derived type and TYPE.

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 **OPTIONS** or **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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Derived type, TYPE, MAP...END MAP, RECORD, UNION...END UNION, PACK Directive, OPTIONS Directive, Data Types, Constants, and Variables, Record Structures

Examples

An item can be a **RECORD** statement that references a previously defined structure type:

```
STRUCTURE /full_address/
  RECORD  /full_name/ personsname
  RECORD  /address/   ship_to
  INTEGER*1           age
  INTEGER*4           phone
END STRUCTURE
```

You can specify a particular item by listing the sequence of items required to reach it, separated by a period (.). Suppose you declare a structure variable, `shippingaddress`, using the `full_address` structure defined in the previous example:

```
RECORD /full_address/ shippingaddress
```

In this case, the `age` item would then be specified by `shippingaddress.age`, the first name of the receiver by `shippingaddress.personsname.first_name`, and so on.

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`.

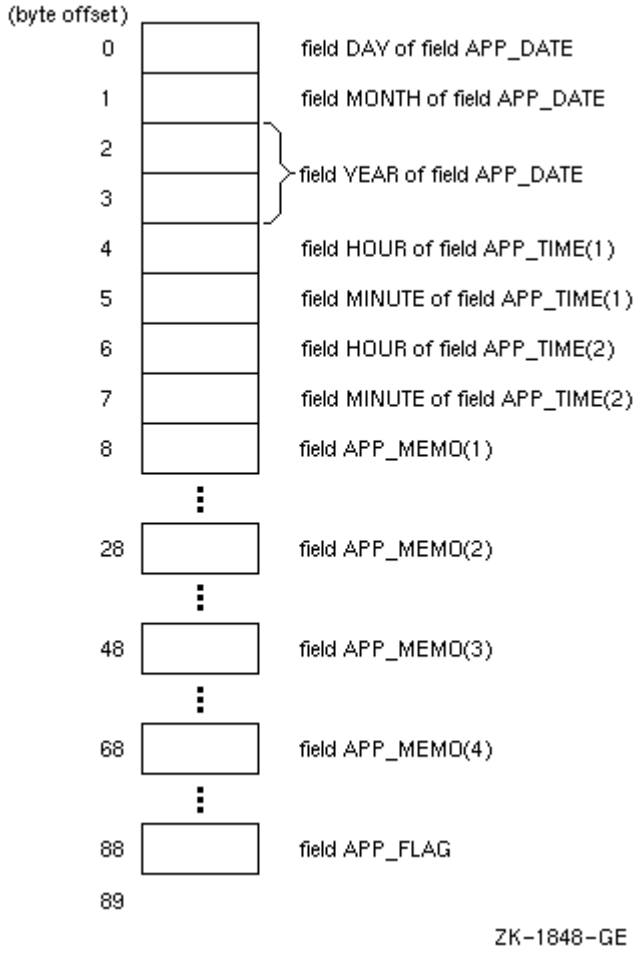
```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

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.

The following figure shows the memory mapping of any record or record array element with the structure `APPOINTMENT`.

Memory Map of Structure APPOINTMENT



SUBROUTINE

Statement: The initial statement of a subroutine subprogram. A subroutine subprogram is invoked in a **CALL** statement or by a defined assignment statement, and does not return a particular value.

Syntax

[*prefix*] **SUBROUTINE** *name* [([*d-arg-list*])]

prefix
(Optional) Is one of the following:

type [*keyword*]
keyword [*type*]

type
Is a data type specifier.

keyword
Is **RECURSIVE**, **PURE**, or **ELEMENTAL**.

The keyword **RECURSIVE** indicates a recursive subroutine, which can reference itself directly or indirectly.

The keyword PURE asserts that the procedure has no side effects. The keyword ELEMENTAL indicates a restricted form of pure procedure.

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.

You need an interface block for a subroutine when:

- Calling arguments use argument keywords.

- Some arguments are optional.
- A dummy argument is an assumed-shape array, a pointer, or a target.
- The subroutine extends intrinsic assignment.
- The subroutine can be referenced by a generic name.
- The subroutine is in a dynamic-link library.

If the subroutine is in a DLL and is called from your program, use the option DLLEXPORT or DLLIMPORT, which you can specify with the ATTRIBUTES directive.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FUNCTION, INTERFACE, PURE, ELEMENTAL, CALL, RETURN, ENTRY, Argument Association, Program Units and Procedures, General Rules for Function and Subroutine Subprograms, Obsolescent and Deleted Language Features

Examples

The following example shows a subroutine:

```
Main Program          Subroutine
CALL HELLO_WORLD      SUBROUTINE HELLO_WORLD
...                   PRINT *, "Hello World"
END                   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)             SUBROUTINE CHECK(X,Y,*,*,Q)
    TYPE *, 'VALUE LESS THAN ZERO'        ...
    GO TO 30                          50  IF (Z)  60,70,80
10  TYPE*, 'VALUE EQUALS ZERO'        60  RETURN
    GO TO 30                          70  RETURN 1
20  TYPE*, 'VALUE MORE THAN ZERO'     80  RETURN 2
30  CONTINUE                              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 90 and Fortran 95.)

The following shows another example:

```
     SUBROUTINE GetNum (num, unit)
     INTEGER num, unit
 10  READ (unit, '(I10)', ERR = 10) num
     END
```

SUBTITLE

Compiler Directive: Specifies a string for the subtitle field of a listing header.

Syntax

*c***DEC$ SUBTITLE** *string*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

string
Is a character constant containing up to 31 printable characters.

Rules and Behavior

To enable the **SUBTITLE** directive, you must specify the compiler option that produces a source listing file.

When **SUBTITLE** appears on a page of a listing file, the specified string appears in the listing header of the following page.

If the directive appears more than once on a page, the last directive is the one in effect for the following page.

If the directive does not specify a string, no change occurs in the listing file header.

The following form is also allowed: `!MS$SUBTITLE:string`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: TITLE, MESSAGE, General Compiler Directives

Example

```
!DEC$ TITLE:'Program MATHSTAT'
REAL epsilon, delta
INTEGER i1, i2, i3
```

```
CALL STAT(epsilon, delta)
CALL MATH (i1, i2, i3)
END
SUBROUTINE STAT(a, b)
  !DEC$ SUBTITLE:'Subroutine STAT'
  REAL a, b
  CALL statpack(a, b)
  !DEC$ SUBTITLE:''
END SUBROUTINE STAT

SUBROUTINE MATH(a, b, c)
  !DEC$ SUBTITLE:'Subroutine MATH'
  INTEGER a, b, c
  a = b * c
  !DEC$ SUBTITLE:''
END SUBROUTINE MATH
```

SUM

Transformational Intrinsic Function (Generic): Returns the sum of all the elements in an entire array or in a specified dimension of an array.

Syntax

result = **SUM** (*array* [, *dim*] [, *mask*])

array
(Input) Must be an array of type integer, real, or complex.

dim
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.

mask
(Optional; input) 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 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 there are no true elements, 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)].

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PRODUCT

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

```
[ 1  2  3 ]
[ 4  5  6 ].
```

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.

The following shows another example:

```
INTEGER array (2, 3), i, j(3)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  array is  1 3 5
!            2 4 6
i = SUM((/ 1, 2, 3 /))        ! returns 6
j = SUM(array, DIM = 1)       ! returns [3 7 11]
WRITE(*,*) i, j
END
```

SYSTEM

Portability Function: Sends a command to the shell as if it had been typed at the command line.

Module: USE DFPORT

Syntax

result = **SYSTEM** (*string*)

string
(Input) Character*(*). Operating system command.

Results:

The result type is INTEGER(4). The result is the exit status of the shell command. If -1, use IERRNO to retrieve the error. Errors can be one of the following:

- **E2BIG**: The argument list is too long.
- **ENOENT**: The command interpreter cannot be found.
- **ENOEXEC**: The command interpreter file has an invalid format and is not executable.
- **ENOMEM**: Not enough system resources are available to execute the command.

The calling process waits until the command terminates.

Commands run with the **SYSTEM** routine are run in a separate shell. Defaults set with the **SYSTEM** function, such as current working directory or environment variables, do not affect the environment the calling program runs in.

The command line character limit for the **SYSTEM** function is the same limit that your operating system command interpreter accepts.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SYSTEMQQ

Example

```
USE DFPORT
INTEGER(4) I, errnum
I = SYSTEM("dir > file.lst")
If (I .eq. -1) then
  errnum = ierrno( )
  print *, 'Error ', errnum
end if
END
```

SYSTEM_CLOCK

Intrinsic Subroutine: Returns integer data from a real-time clock.

Syntax

CALL SYSTEM_CLOCK ([*count*] [, *count_rate*] [, *count_max*])

count
(Optional; output) 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
(Optional; output) Must be scalar and of type default integer. It is set to the number of processor clock counts per second.

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
(Optional; output) Must be scalar and of type default integer. It is set to the maximum value that *count* can have, **HUGE**(0).

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME, HUGE, GETTIM

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
```

SYSTEMQQ

Run-Time Function: Executes a system command by passing a command string to the operating system's command interpreter.

Module: USE DFLIB

Syntax

result = **SYSTEMQQ** (*commandline*)

commandline
(Input) Character*(*). Command to be passed to the operating system.

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The **SYSTEMQQ** function allows you to pass operating-system commands as well as programs. **SYSTEMQQ** refers to the COMSPEC and PATH environment variables that locate the command interpreter file (usually named COMMAND.COM).

If the function fails, call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

- **ERR$2BIG**: The argument list exceeds 128 bytes, or the space required for the environment formation exceeds 32K.
- **ERR$NOINT**: The command interpreter cannot be found.
- **ERR$NOEXEC**: The command interpreter file has an invalid format and is not executable.
- **ERR$NOMEM**: Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

The command line character limit for the **SYSTEMQQ** function is the same limit that your operating system command interpreter accepts.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SYSTEM

Example

```
USE DFLIB
LOGICAL(4) result
result = SYSTEMQQ('copy c:\bin\fmath.dat &
                        c:\dat\fmath2.dat')
```

TAN

Elemental Intrinsic Function (Generic): Produces a tangent (with the result in radians).

Syntax

result = **TAN** (*x*)

x
(Input) Must be of type real. It must be in radians and is treated as modulo 2 * pi.

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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

TAN (2.0) has the value -2.185040.

TAN (0.8) has the value 1.029639.

TAND

Elemental Intrinsic Function (Generic): Produces a tangent (with the result in degrees).

Syntax

result = **TAND** (*x*)

x
(Input) 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)
DTAND	REAL(8)	REAL(8)
QTAND [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

TAND (2.0) has the value 3.4920771E-02.

TAND (0.8) has the value 1.3963542E-02.

TANH

Elemental Intrinsic Function (Generic): Produces a hyperbolic tangent.

Syntax

result = **TANH** (*x*)

x
(Input) 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 [1]	REAL(16)	REAL(16)
[1] VMS and U*X		

Examples

TANH (2.0) has the value 0.9640276.

TANH (0.8) has the value 0.6640368.

TARGET

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **TARGET** [, *att-ls*] **::** *object* [(*a-spec*)] [, *object* [(*a-spec*)]]...

Statement:

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
(Optional) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALLOCATE, ASSOCIATED, POINTER, Pointer Assignments, Pointer Association, Type Declarations, Compatible attributes.

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
```

The following fragment is from the program POINTER2.F90 in the \DF\SAMPLES\TUTORIAL subdirectory.

```
 ! An example of pointer assignment.
     REAL, POINTER :: arrow1 (:)
     REAL, POINTER :: arrow2 (:)
     REAL, ALLOCATABLE, TARGET :: bullseye (:)

     ALLOCATE (bullseye (7))
     bullseye = 1.
     bullseye (1:7:2) = 10.
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

     arrow1 => bullseye
     WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1
     . . .
```

TIME

TIME can be used as an intrinsic subroutine or as a portability routine.

TIME Intrinsic Subroutine

Intrinsic Subroutine: Returns the current time as set within the system.

Syntax

CALL TIME (*buf*)

buf
Is a 8-byte variable, array, array element, or character substring.

The date is returned as a 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 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
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.

TIME Portability Routine

Portability Function and Subroutine: The function returns the system time, in seconds, since 00:00:00 Greenwich mean time, January 1, 1970. The subroutine fills a parameter with the current time as a string in the format hh:mm:ss.

Module: USE DFPORT

Function Syntax

result = **TIME** ( )

Subroutine Syntax

CALL TIME (*string*)

string
(Output) Character*(*). Current time, based on a 24-hour clock, in the form hh:mm:ss, where hh, mm, and ss are two-digit representations of the current hour, minutes past the hour, and seconds past the minute, respectively.

Results:

The result type is INTEGER(4). The result is the number of seconds that have elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

The value returned by this function is used as input to other Portability date and time functions.

You can use both the function and subroutine versions of **TIME** only if your program includes the

USE DFPORT statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: DATE_AND_TIME

Example

```
USE DFPORT
INTEGER(4) int_time
character*8 char_time
int_time = TIME( )
call TIME(char_time)
print *, 'Integer: ', int_time, 'time: ', char_time
END
```

TIMEF

Portability Function: Returns the number of seconds since the first time it is called, or zero.

Module: USE DFPORT

Syntax

result = **TIMEF** ( )

Results:

The result type is REAL(8). The result is the number of seconds that have elapsed since the first time **TIMEF** was called. The first time called, **TIMEF** returns 0.0D0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Date and Time Procedures

Example

```
USE DFPORT
INTEGER i, j
REAL(8) elapsed_time
elapsed_time = TIMEF( )
DO i = 1, 100000
   j = j + 1
END DO
elapsed_time = TIMEF( )
PRINT *, elapsed_time
END
```

TINY

Inquiry Intrinsic Function (Generic): Returns the smallest number in the model representing the same type and kind parameters as the argument.

Syntax

result = **TINY** (*x*)

x
(Input) Must be of type real; it can be scalar or array valued.

Results:

The result type is 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 Model for Real Data.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: HUGE, Data Representation Models

Examples

If X is of type REAL(4), TINY (X) has the value 2^{-126}.

The following shows another example:

```
REAL(8) r, result
r = 487923.3D0
result = TINY(r)   ! returns 2.225073858507201E-308
```

TITLE

Compiler Directive: Specifies a string for the title field of a listing header.

Syntax

*c***DEC$ TITLE** *string*

c
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

string
Is a character constant containing up to 31 printable characters.

Rules and Behavior

To enable the **TITLE** directive, you must specify the compiler option that produces a source listing file.

When **TITLE** appears on a page of a listing file, the specified string appears in the listing header of the following page.

If the directive appears more than once on a page, the last directive is the one in effect for the following page.

If the directive does not specify a string, no change occurs in the listing file header.

The following form is also allowed: `!MS$TITLE:string`

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SUBTITLE, General Compiler Directives.

Example

```
!DEC$ TITLE:'Program MATHSTAT Version 3.0 9/02/96'
INTEGER i, j, k
REAL a, b, c
CALL hilbert(i, j, k)
CALL erf(a, b, c)
END
```

TRAILZ

Elemental Intrinsic Function: Returns the number of trailing zero bits in an integer.

Syntax

result = **TRAILZ** (*i*)

i
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 Model for Bit Data.

Example

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
```

TRANSFER

Transformational Intrinsic Function (Generic): Converts the bit pattern of the first argument according to the type and kind parameters of the second argument.

Syntax

result = **TRANSFER** (*source*, *mold* [, *size*])

source
(Input) Must be a scalar or array (of any data type).

mold
(Input) Must be a scalar or array (of any data type). It provides the type characteristics (not a value) for the result.

size
(Optional; input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

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).

The following shows another example:

```
COMPLEX CVECTOR(2), CX(1)
!  The next statement sets CVECTOR to
!  [ 1.1 + 2.2i, 3.3 + 0.0i ]
CVECTOR = TRANSFER((/1.1, 2.2, 3.3, 0.0/), &
                   (/(0.0, 0.0)/))
!  The next statement sets CX to [ 1.1 + 2.2i ]
CX = TRANSFER((/1.1, 2.2, 3.3/) , (/(0.0, 0.0)/), &
              SIZE= 1)
WRITE(*,*) CVECTOR
WRITE(*,*) CX
END
```

TRANSPOSE

Transformational Intrinsic Function (Generic): Transposes an array of rank two.

Syntax

result = **TRANSPOSE** (*matrix*)

matrix
(Input) 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: RESHAPE, PRODUCT

Examples

B is the array

```
[ 2  3  4 ]
[ 5  6  7 ]
[ 8  9  1 ].
```

TRANSPOSE (B) has the value

```
[ 2  5  8 ]
[ 3  6  9 ]
[ 4  7  1 ].
```

The following shows another example:

```
INTEGER array(2, 3), result(3, 2)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
! array is  1  3  5
!           2  4  6
result = TRANSPOSE(array)
! result is 1  2
!           3  4
!           5  6
END
```

TRIM

Transformational Intrinsic Function (Generic): Returns the argument with trailing blanks removed.

Syntax

result = **TRIM** (*string*)

string
(Input) Must be a scalar of type character.

Results:

The result type is 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LEN_TRIM

Examples

In these examples, the symbol - represents a blank.

TRIM (’--NAME----’) has the value ’--NAME’.

TRIM (’--C--D-----’) has the value ’--C--D’.

The following shows another example:

```
! next line prints 28
WRITE(*, *) LEN("I have blanks behind me       ")
! the next line prints 23
WRITE(*,*) LEN(TRIM("I have blanks behind me       "))
END
```

TYPE

Statement: Declares a variable to be a derived type. For more information, see Derived Type.

Example

```
!   DERIVED.F90
!   Define a derived-type structure,
!   type variables, and assign values

    TYPE member
      INTEGER age
      CHARACTER (LEN = 20) name
    END TYPE member

    TYPE (member) :: george
    TYPE (member) :: ernie

    george     = member( 33, ’George Brown’ )
    ernie%age  = 56
    ernie%name = ’Ernie Brown’

    WRITE (*,*) george
    WRITE (*,*) ernie
    END
```

Type Declarations

Statement: Explicitly specifies the properties of data objects or functions.

Syntax

A type declaration statement has the general form:

type [[, *att*] ... **::**] *v* [/*c-list*/] [, *v* [/*c-list*/]]...

type
Is one of the following data type specifiers:

BYTE
INTEGER [*kind-selector*]
REAL [*kind-selector*]
DOUBLE PRECISION
COMPLEX [*kind-selector*]
DOUBLE COMPLEX
CHARACTER [*char-selector*]
LOGICAL [*kind-selector*]
TYPE (*derived-type-name*)

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.

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.

See each data type for further information on that type.

att
Is one of the following attribute specifiers:

ALLOCATABLE	INTRINSIC	PUBLIC [1]
AUTOMATIC	OPTIONAL	SAVE
DIMENSION	PARAMETER	STATIC
EXTERNAL	POINTER	TARGET
INTENT	PRIVATE [1]	VOLATILE
[1] These are access specifiers.		

You can also declare any attribute separately as a statement.

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:

- A dummy argument
- A function result
- An object in a named common block (unless the type declaration is in a block data program unit)
- An object in blank common
- An allocatable array
- An external name
- An intrinsic name
- An automatic object
- An object that has the AUTOMATIC attribute

An object can have more than one attribute. The following table lists the compatible attributes:

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
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	

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: CHARACTER, COMPLEX, Derived Type, DOUBLE COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, REAL, IMPLICIT, RECORD, STRUCTURE...END STRUCTURE, TYPE, Type Declaration Statements

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()
```

The following shows more examples:

```
 REAL a (10)
 LOGICAL, DIMENSION (5, 5) :: mask1, mask2
 COMPLEX :: cube_root = (-0.5, 0.867)
 INTEGER, PARAMETER :: short = SELECTED_INT_KIND (4)
 REAL (KIND (0.0D0)) a1
 REAL (KIND = 2) b
 COMPLEX (KIND = KIND (0.0D0)) :: c
 INTEGER (short) k ! Range at least -9999 to 9999
 TYPE (member) :: george
```

UBOUND

Inquiry Intrinsic Function (Generic): Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.

Syntax

result = **UBOUND** (*array* [, *dim*])

array
(Input) 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
(Optional; input) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.

Results:

The result type is default integer. 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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: LBOUND

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.

The following shows another example:

```
REAL ar1(2:3, 4:5, -1:14), vec1(35)
INTEGER res1(3), res2, res3(1)
res1 = UBOUND (ar1)          ! returns [3, 5, 14]
res2 = UBOUND (ar1, DIM= 3)  ! returns 14
res3 = UBOUND (vec1)         ! returns 35
END
```

UNION...END UNION

Statements: Define 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.

Syntax

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. It can be of any intrinsic or derived type.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: STRUCTURE...END STRUCTURE, Record Structures

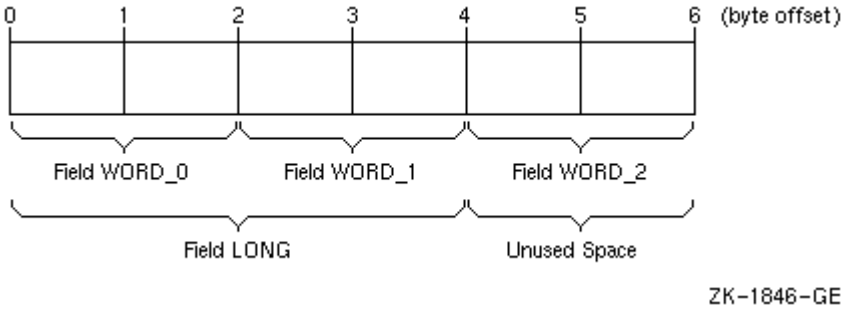
Examples

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. The following figure shows the memory mapping of any record with the structure WORDS_LONG:

Memory Map of Structure WORDS_LONG



In the following example, note how the first 40 characters in the string2 array are overlayed on 4-byte integers, while the remaining 20 are overlayed on 2-byte integers:

```
  UNION
     MAP
       CHARACTER*20 string1, CHARACTER*10 string2(6)
     END MAP
     MAP
       INTEGER*2 number(10), INTEGER*4 var(10), INTEGER*2
+       datum(10)
     END MAP
  END UNION
```

UNLINK

Portability Function: Deletes the file given by path.

Module: USE DFPORT

Syntax

result = **UNLINK** (*name*)

name
(Input) Character*(*). Path of the file to delete. The path can use forward (/) or backward (\) slashes as path separators and can contain drive letters.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code. Errors can be one of the following:

ENOENT: The specified file could not be found.

EACCES: The specified file is read-only.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SYSTEM, DELDIRQQ

Example

```
USE DFPORT
INTEGER(4) ISTATUS
CHARACTER*20 dirname
READ *, dirname
ISTATUS = UNLINK (dirname)
IF (ISTATUS) then
   print *, 'Error ', ISTATUS
END IF
END
```

UNLOCK

Statement: Frees a record in a relative or sequential file that was locked by a previous **READ** statement.

Syntax

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.

See Also: Data Transfer I/O Statements, Branch Specifiers

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`.

UNPACK

Transformational Intrinsic Function (Generic): Takes elements from a rank-one array and unpacks them into another (possibly larger) array under the control of a mask.

Syntax

result = **UNPACK** (*vector, mask, field*)

vector
(Input) 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
(Input) Must be a logical array. It determines where elements of *vector* are placed when they are unpacked.

field
(Input) 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).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PACK, RESHAPE, SHAPE

Examples

N is the array

```
[ 0  0  1 ]
[ 1  0  1 ]
[ 1  0  0 ],
```

P is the array (2, 3, 4, 5), and Q is the array

```
[ T  F  F ]
[ F  T  F ]
[ T  T  F ].
```

UNPACK (P, MASK=Q, FIELD=N) produces the result

```
[ 2  0  1 ]
[ 1  4  1 ]
[ 3  5  0 ].
```

UNPACK (P, MASK=Q, FIELD=1) produces the result

```
[ 2  1  1 ]
[ 1  4  1 ]
[ 3  5  1 ].
```

The following shows another example:

```
LOGICAL mask (2, 3)
INTEGER vector(3) /1, 2, 3/, AR1(2, 3)
mask = RESHAPE((/.TRUE.,.FALSE.,.FALSE.,.TRUE.,&
                .TRUE.,.FALSE./), (/2, 3/))
! vector = [1 2 3] and mask =  T F T
!                              F T F
AR1 = UNPACK(vector, mask, 8)  ! returns  1 8 3
                               !          8 2 8
END
```

UNPACKTIMEQQ

Run-Time Subroutine: Unpacks a packed time and date value into its component parts.

Module: USE DFLIB

Syntax

CALL UNPACKTIMEQQ (*timedate*, *iyr*, *imon*, *iday*, *ihr*, *imin*, *isec*)

timedate
(Input) INTEGER(4). Packed time and date information.

iyr
(Output) INTEGER(2). Year (*xxxx* AD).

imon
(Output) INTEGER(2). Month (1 - 12).

iday
(Output) INTEGER(2). Day (1 - 31).

ihr
(Output) INTEGER(2). Hour (0 - 23).

imin
(Output) INTEGER(2). Minute (0 - 59).

isec
(Output) INTEGER(2). Second (0 - 59).

GETFILEINFOQQ returns time and date in a packed format. You can use **UNPACKTIMEQQ** to unpack these values. Use **PACKTIMEQQ** to repack times for passing to **SETFILETIMEQQ**. Packed times can be compared using relational operators.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: PACKTIMEQQ, GETFILEINFOQQ

Example

```
USE DFLIB
CHARACTER(80) file
TYPE (FILE$INFO) info
INTEGER(4) handle, result
INTEGER(2) iyr, imon, iday, ihr, imin, isec

file = 'd:\f90ps\bin\t???.*'
handle = FILE$FIRST
result = GETFILEINFOQQ(file, info, handle)
CALL UNPACKTIMEQQ(info.lastwrite, iyr, imon,&
                  iday, ihr,imin, isec)
WRITE(*,*) iyr, imon, iday
WRITE(*,*) ihr, imin, isec
END
```

UNREGISTERMOUSEEVENT

QuickWin Function: Removes the callback routine registered for a specified window by an earlier call to **REGISTERMOUSEEVENT**.

Module: USE DFLIB

Syntax

result = **UNREGISTERMOUSEEVENT** (*unit*, *mouseevents*)

unit

(Input) INTEGER(4). Unit number of the window whose callback routine on mouse events is to be unregistered.

mouseevents
(Input) INTEGER(4). One or more mouse events handled by the callback routine to be unregistered. Symbolic constants (defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory) for the possible mouse events are:

- **MOUSE$LBUTTONDOWN**: Left mouse button down
- **MOUSE$LBUTTONUP**: Left mouse button up
- **MOUSE$LBUTTONDBLCLK**: Left mouse button double-click
- **MOUSE$RBUTTONDOWN**: Right mouse button down
- **MOUSE$RBUTTONUP**: Right mouse button up
- **MOUSE$RBUTTONDBLCLK**: Right mouse button double-click
- **MOUSE$MOVE**: Mouse moved

Results:

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer which can be one of the following:

- **MOUSE$BADUNIT**: The unit specified is not open, or is not associated with a QuickWin window.
- **MOUSE$BADEVENT**: The event specified is not supported.

Once you call **UNREGISTERMOUSEEVENT**, QuickWin no longer calls the callback routine specified earlier for the window when mouse events occur. Calling **UNREGISTERMOUSEEVENT** when no callback routine is registered for the window has no effect.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, REGISTERMOUSEEVENT, WAITONMOUSEEVENT

USE

Statement: Gives a program unit accessibility to public entities in a module.

Syntax

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Program Units and Procedures, USE Statement (more examples)

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:

```
MODULE 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.

The following shows another example:

```
! Module containing original type declarations
MODULE geometry
type square
   real side
   integer border
end type
type circle
   real radius
   integer border
end type
END MODULE

! Program renames module types for local use.
PROGRAM test
USE GEOMETRY,LSQUARE=>SQUARE,LCIRCLE=>CIRCLE
! Now use these types in declarations
type (LSQUARE) s1,s2
type (LCIRCLE) c1,c2,c3
```

%VAL

Built-in Function: Changes the form of an actual argument. Passes the argument as an immediate value.

Syntax

result = **%VAL** (*a*)

a
(Input) An expression, record name, procedure name, array, character array section, or array element.

The argument is passed as follows:

- On Intel processors, as a 32-bit immediate value. If the argument is integer (or logical) and shorter that 32 bits, it is sign-extended to a 32-bit value. For complex data types, **%VAL** passes two 32-bit arguments.
- On Alpha processors, as a 64-bit immediate value. If the argument is integer (or logical) and shorter that 64 bits, it is sign-extended to a 64-bit value. For complex data types, **%VAL** passes two 64-bit arguments.

You must specify **%VAL** in the actual argument list of a **CALL** statement or function reference. You cannot use it in any other context.

The following table lists the DIGITAL Fortran defaults for argument passing, and the allowed uses of **%VAL**:

Actual Argument Data Type	**Default**	**%VAL**
Expressions:		
Logical	REF	Yes[1]
Integer	REF	Yes[1]
REAL(4)	REF	Yes
REAL(8)	REF	Yes[2]
REAL(16) [3]	REF	No
COMPLEX(4)	REF	No
COMPLEX(8)	REF	No
Character	See table note[4]	No
Hollerith	REF	No
Aggregate[5]	REF	No
Derived	REF	No
Array Name:		
Numeric	REF	No
Character	See table note[4]	No
Aggregate[5]	REF	No
Derived	REF	No
Procedure Name:		
Numeric	REF	No
Character	See table note[4]	No

[1] If a logical or integer value occupies less than 64 (Alpha systems) or 32 (Intel systems) bits of

storage, it is converted to the correct size by sign extension. Use the ZEXT intrinsic function if zero extension is desired.
[2] Alpha only
[3] VMS, U*X
[4] On DIGITAL UNIX, Windows NT and Windows 95 systems, a character argument is passed by address and hidden length.
[5] In DIGITAL Fortran record structures

See Also: CALL, %REF

Example

```
CALL SUB(2, %VAL(2))
```

Constant 2 is passed by reference. The second constant 2 is passed by immediate value.

VERIFY

Elemental Intrinsic Function (Generic): 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.

Syntax

result = **VERIFY** (*string*, *set* [, *back*])

string
(Input) Must be of type character.

set
(Input) Must be of type character with the same kind parameter as *string*.

back
(Optional; input)Must be of type logical.

Results:

The result type is default integer.

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SCAN

Examples

VERIFY (’CDDDC’, ’C’) has the value 2.

VERIFY (’CDDDC’, ’C’, BACK=.TRUE.) has the value 4.

VERIFY (’CDDDC’, ’CD’) has the value zero.

The following shows another example:

```
INTEGER(4) position

position = VERIFY (’banana’, ’nbc’)  ! returns 2
position = VERIFY (’banana’, ’nbc’, BACK=.TRUE.)
                                     ! returns 6
position = VERIFY (’banana’, ’nbca’) ! returns 0
```

VIRTUAL

Statement: Has the same form and effect as the DIMENSION statement. It is included for compatibility with PDP-11 FORTRAN.

VOLATILE

Statement and 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:

Syntax

Type Declaration Statement:

type, [*att-ls*,] **VOLATILE** [, *att-ls*] **::** *object* [, *object*]...

Statement:

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**.

Formal (dummy) arguments that can be omitted 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:

- A procedure
- A function result
- A namelist group

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: Type Declarations, Compatible attributes.

Examples

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(KIND=1) IPI(4)
INTEGER(KIND=4) A, B, C, D, E, ILOOK
```

```
INTEGER(KIND=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.

WAITONMOUSEEVENT

QuickWin Function: Waits for the specified mouse input from the user.

Module: USE DFLIB

Syntax

result = **WAITONMOUSEEVENT** (*mouseevents*, *keystate*, *x*, *y*)

mouseevents
(Input) INTEGER(4). One or more mouse events that must occur before the function returns. Symbolic constants for the possible mouse events are:

- **MOUSE$LBUTTONDOWN**: Left mouse button down
- **MOUSE$LBUTTONUP**: Left mouse button up
- **MOUSE$LBUTTONDBLCLK**: Left mouse button double-click
- **MOUSE$RBUTTONDOWN**: Right mouse button down
- **MOUSE$RBUTTONUP**: Right mouse button up
- **MOUSE$RBUTTONDBLCLK**: Right mouse button double-click
- **MOUSE$MOVE**: Mouse moved

keystate
(Output) INTEGER(4). Bitwise inclusive OR of the state of the mouse duing the event. The value returned in *keystate* can be any or all of the following symbolic constants:

- **MOUSE$KS_LBUTTON**: Left mouse button down during event
- **MOUSE$KS_RBUTTON**: Right mouse button down during event
- **MOUSE$KS_SHIFT**: SHIFT key held down during event
- **MOUSE$KS_CONTROL**: CONTROL key held down during event

x
(Output) INTEGER(4). X position of the mouse when the event occurred.

y
(Output) INTEGER(4). Y position of the mouse when the event occurred.

Results:

The result type is INTEGER(4). The result is the symbolic constant associated with the mouse event

that occurred if successful. If the function fails, it returns the constant MOUSE$BADEVENT, meaning the event specified is not supported.

WAITONMOUSEEVENT does not return until the specified mouse input is received from the user. While waiting for a mouse event to occur, the status bar changes to read "Mouse input pending in XXX" where XXX is the name of the window. When a mouse event occurs, the status bar returns to its previous value.

A mouse event must happen in the window that had focus when **WAITONMOUSEEVENT** was initially called. Mouse events in other windows will not end the wait. Mouse events in other windows cause callbacks to be called for the other windows, if callbacks were previously registered for those windows.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: REGISTERMOUSEEVENT, UNREGISTERMOUSEEVENT, Using QuickWin

WHERE

Statement and Contruct: Lets 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.

Syntax

Statement:

WHERE (*mask-expr1*) *assign-stmt*

Contruct:

[*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*
- 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 shows an example using 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*".

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.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: FORALL, Arrays

Examples

```
REAL(4) a(20)
. . .
WHERE (a > 0.0)
  a = LOG (a)
     !LOG is invoked only for positive elements
END WHERE
```

WRAPON

Graphics Function: Controls whether text output with the **OUTTEXT** function wraps to a new line or is truncated when the text output reaches the edge of the defined text window.

Module: USE DFLIB

Syntax

result = **WRAPON** (*option*)

option
(Input) INTEGER(2). Wrap mode. One of the following symbolic constants:

- **$GWRAPOFF**: Truncates lines at right edge of window border.
- **$GWRAPON**: Wraps lines at window border, scrolling if necessary.

Results:

The result type is INTEGER(2). The result is the previous value of *option*.

WRAPON does not affect font routines such as **OUTGTEXT**.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: OUTTEXT, SCROLLTEXTWINDOW, SETTEXTPOSITION, SETTEXTWINDOW

Example

```
USE DFLIB
INTEGER(2) row, status2
INTEGER(4) status4
TYPE ( rccoord ) curpos
TYPE ( windowconfig ) wc
```

```
LOGICAL status

status = GETWINDOWCONFIG( wc )
wc.numtextcols = 80
wc.numxpixels  = -1
wc.numypixels  = -1
wc.numtextrows = -1
wc.numcolors   = -1
wc.fontsize    = -1
wc.title = "This is a test"C
wc.bitsperpixel = -1
status = SETWINDOWCONFIG( wc )
status4= SETBKCOLORRGB(#FF0000 )
CALL CLEARSCREEN( $GCLEARSCREEN )

!  Display wrapped and unwrapped text in text windows.
CALL SETTEXTWINDOW( INT2(1),INT2(1),INT2(5),INT2(25))
CALL SETTEXTPOSITION(INT2(1),INT2(1), curpos )
status2 = WRAPON( $GWRAPON )
status4 = SETTEXTCOLORRGB(#00FF00)
DO i = 1, 5
   CALL OUTTEXT( 'Here text does wrap. ')
END DO
CALL SETTEXTWINDOW(INT2(7),INT2(10),INT2(11),INT2(40))
CALL SETTEXTPOSITION(INT2(1),INT2(1),curpos)
status2 = WRAPON( $GWRAPOFF )
status4 = SETTEXTCOLORRGB(#008080)
DO row = 1, 5
   CALL SETTEXTPOSITION(INT2(row), INT2(1), curpos )
   CALL OUTTEXT('Here text does not wrap. ')
   CALL OUTTEXT('Here text does not wrap.')
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

WRITE

Statement: Transfers output data to external sequential, direct-access, or internal records.

Syntax

Sequential

Formatted

WRITE (*eunit*, *format* [, *advance*] [, *iostat*] [, *err*]) [*io-list*]

Formatted: List-Directed

WRITE (*eunit*, * [, *iostat*] [, *err*]) [*io-list*]

Formatted: Namelist

WRITE (*eunit*, *nml-group* [, *iostat*] [, *err*])

Unformatted

WRITE (*eunit* [, *iostat*] [, *err*]) [*io-list*]

Direct-Access

Formatted

WRITE (*eunit, format, rec* [, *iostat*] [, *err*]) [*io-list*]

Unformatted

WRITE (*eunit, rec* [, *iostat*] [, *err*]) [*io-list*]

Indexed (VMS only)

Formatted

READ (*eunit, format,* [,*iostat*] [,*err*]) [*io-list*]

Unformatted

READ (*eunit,* [,*iostat*] [,*err*]) [*io-list*]

Internal

WRITE (*iunit, format* [, *iostat*] [, *err*]) [*io-list*]

eunit
Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.

format
Is a format specifier. It is optionally prefaced by FMT= if fo*rmat* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=.

For internal **WRITE**s, an asterisk (*) indicates list-directed formatting. For direct-access **WRITE**s, an asterisk is not permitted.

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'.

iostat
Is the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.

err
Are branch specifiers if an error (ERR=label) condition occurs.

io-list
Is an I/O list: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-**DO** list.

form
Is the nonkeyword form of a format specifier (no FMT=).

*
Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml-group
Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier.

rec
Is the cell number of a record to be accessed directly. Optionally prefaced by REC=.

iunit
Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list.

It must be a character variable. It must not be an array section with a vector subscript.

If a parameter of the **WRITE** statement is an expression that calls a function, that function must not execute an I/O statement or the **EOF** intrinsic function, because the results are unpredictable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: I/O Lists, I/O Control List, Forms for Sequential WRITE Statements, Forms for Direct-Access WRITE Statements, Forms for Indexed WRITE Statements (VMS only), Forms and Rules for Internal WRITE Statements, READ, PRINT, OPEN, I/O Formatting

Example

```
! write to file
open(1,FILE='test.dat')
write (1, '(A20)') namedef
! write with FORMAT statement
WRITE (*, 10) (n, SQRT(FLOAT(n)), FLOAT(n)**(1.0/3.0), n = 1, 100)
10 FORMAT (I5, F8.4, F8.5)
!
WRITE(6,'("Expected ",F12.6)') 2.0
```

XOR

Elemental Intrinsic Function: See IEOR.

Example

```
INTEGER i, j, k
i = 3              ! 011
j = 5              ! 101
k = XOR(i, j)      ! returns 6 = 110
```

ZEXT

Elemental Intrinsic Function (Generic): Extends the argument with zeros. This function is used primarily for bit-oriented operations.

Syntax

result = **ZEXT** (*x*)

x
(Input) Must be of type logical or integer.

Results:

The result type is default integer. 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 option /integer_size can affect **ZEXT**.

Specific Name	Argument Type	Result Type
IZEXT	LOGICAL(1)	INTEGER(2)
	LOGICAL(2)	INTEGER(2)
	INTEGER(1)	INTEGER(2)
	INTEGER(2)	INTEGER(2)
JZEXT	LOGICAL(1)	INTEGER(4)
	LOGICAL(2)	INTEGER(4)
	LOGICAL(4)	INTEGER(4)
	INTEGER(1)	INTEGER(4)
	INTEGER(2)	INTEGER(4)

	INTEGER(4)	INTEGER(4)
KZEXT [1]	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)

[1] Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

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.

Glossary

A - B - C - D - E - F - G - H - I - K - L - M - N - O - P - Q - R - S - T - U - V - W

A

absolute pathname
On DIGITAL UNIX, Windows NT, and Windows 95 systems, a directory path specified in fixed relationship to the root directory. On DIGITAL UNIX systems, the first character is a slash (/). On Windows NT and Windows 95 systems, the first character is a backslash (\).

active screen buffer
The screen buffer that is currently displayed in a console's window.

active window
A top-level window of the application with which the user is working. Windows identifies the active window by highlighting its title bar and border.

actual argument
An expression, variable, procedure, or alternate return specifier which is specified in a subroutine or function reference. The value is passed from the calling program unit to a subprogram.

adjustable array
An explicit-shape array that is a dummy argument to a subprogram. The term is from FORTRAN-77. *See also* explicit-shape array.

aggregate reference
A reference to a record structure field.

allocatable array
A named array that has the ALLOCATABLE attribute. Once space has been allocated for this type of array, the array has a shape and can be defined (and redefined) or referenced. It is an error to allocate an allocatable array that is currently allocated.

allocation status
Indicates whether an allocatable array or pointer is allocated. An allocation status is one of: allocated, deallocated, or undefined. An undefined allocation status means an array can no longer be referenced, defined, allocated, or deallocated. *See also* association status.

alphanumeric
Pertaining to letters and digits.

alternate key
On OpenVMS systems, an optional key within the data records in an indexed file, which can be used to build an alternate index.

alternate return
A subroutine argument that permits control to branch immediately to some position other than the statement following the call. The actual argument in an alternate return is the statement label to which control should be transferred. (An alternate return is an obsolescent feature in Fortran 90.)

ANSI
The American National Standards Institute. An organization through which accredited organizations create and maintain voluntary industry standards.

argument
See actual argument and dummy argument.

argument association

The relationship (or "matching up") between an actual argument and dummy argument during the execution of a procedure reference.

argument keyword

The name of a dummy (formal) argument. It can be used in a procedure reference before the equals sign [keyword = actual argument] provided the procedure has an explicit interface. This association allows actual arguments to appear in any order.

Argument keywords are supplied for many of the intrinsic procedures.

array

A set of scalar data that all have the same type and kind type parameters. An array can be referenced by element (using a subscript), by section (using a section subscript list), or as a whole. An array has a rank (up to 7), bounds, size, and a shape.

An individual array element is a scalar object. An array section, which is itself an array, is a subset of the entire array.

Contrast with scalar. *See also* bounds, conformable, shape, and size.

array constructor

A mechanism used to specify a sequence of scalar values that produce a rank-one array.

To construct an array of rank greater than one, you must apply the RESHAPE intrinsic function to the array constructor.

array element

A scalar item in an array. An array element is identified by the array name followed by one or more subscripts in parentheses, indicating the element's position in the array. For example, B(3) or A(2,5).

array pointer

A pointer to an array. *See also* array and pointer.

array section

A subobject (or portion) of an array. It consists of the set of array elements or substrings of this set. The set (or section subscript list) is specified by subscripts, subscript triplets, and vector subscripts. If the set does not contain at least one subscript triplet or vector subscript, the reference indicates an array element, not an array.

array specification

A program statement specifying an array name and the number of dimensions the array contains (its rank). An array specification can appear in a DIMENSION or COMMON statement, or in a type declaration statement.

ASCII

The American Standard Code for Information Interchange. A 7-bit character encoding scheme associating an integer from 0 through 127 with 128 characters.

assignment

A statement in the form variable = expression. The statement assigns (stores) the value of an expression on the right of an equal sign to the storage location of the variable to the left of the equal sign. In the case of Fortran 90 pointers, the storage location is assigned, not the pointer itself.

association

An assignment of names, pointers, or storage locations which identifies one entity with several names in the same or different scoping units. The principal kinds of association are argument association, host association, pointer association, storage association, and use association.

association status

Indicates whether or not a pointer is associated with a target. An association status is one of: undefined, associated, or disassociated. An undefined association status means a pointer can no longer be referenced, defined, or deallocated. An undefined pointer can, however, be allocated,

nullified, or pointer assigned to a new target. *See also* allocation status.

assumed-length character argument
A dummy argument that assumes the length attribute of the corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

assumed-shape array
A dummy argument array that assumes the shape of its associated actual argument array.

assumed-size array
A dummy array that takes the size of the actual argument passed to it. The rank, extents, and bounds of the dummy array are specified in its declaration, except for the upper bound (which is specified by a *) and the extent of the last dimension.

attribute
A property of a data object that can be specified in a type declaration statement. These properties determine how the data object can be used in a program.

B

background process
On DIGITAL UNIX systems, a process for which the command interpreter is not waiting. Its process group differs from that of its controlling terminal, so it is blocked from most terminal access. *Contrast with* foreground process.

background window
Any window created by a thread other than the foreground thread.

batch process
On OpenVMS systems, a process that runs without user interaction. *Contrast with* interactive process.

big endian
A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the highest addressed byte. *Contrast with* little endian.

binary constant
A constant that is a string of binary (base 2) digits (0 or 1) enclosed by apostrophes or quotation marks and preceded by the letter B.

binary operator
An operator that acts on a pair of operands. The exponentiation, multiplication, division, and concatenation operators are binary operators.

bit constant
A constant that is a binary, octal, or hexadecimal number.

bit field
A contiguous group of bits within a binary pattern; they are specified by a starting bit position and length. Some intrinsic functions (for example, IBSET and BTEST) and the intrinsic subroutine MVBITS operate on bit fields.

bitmap
An array of bits that contains data that describes the colors found in a rectangular region on the screen (or the rectangular region found on a page of printer paper).

blank common
A common block (one or more contiguous areas of storage) without a name. Common blocks are defined by a COMMON statement.

block
A group of statements or constructs that is treated as an integral unit. For example, a block can be a group of constructs or statements that perform a task; the task can be executed once,

repeatedly, or not at all.

block data program unit
A program unit, containing a BLOCK DATA statement and its associated specification statements, that establishes common blocks and assigns initial values to the variables in named common blocks. In FORTRAN 77, this was called a block data subprogram.

bounds
The range of subscript values for elements of an array. The lower bound is the smallest subscript value in a dimension, and the upper bound is the largest subscript value in that dimension. Array bounds can be positive, zero, or negative.
These bounds are specified in an array specification. *See also* array specification.

brush
A bitmap that is used to fill the interior of closed shapes, polygons, ellipses, and paths.

brush origin
A coordinate that specifies the location of one of the pixels in a brush's bitmap. Windows maps this pixel to the upper left corner of the window that contains the object to be painted. *See also* bitmap.

byte-order mark
A special Unicode character (0xFEFF) that is placed at the beginning of Unicode text files to indicate that the text is in Unicode format.

byte reversed
A Unicode file in which the most significant byte is first (as on Motorola architectures).

C

carriage-control character
A character in the first position of a printed record that determines the vertical spacing of the output line.

character constant
A constant that is a string of printable ASCII characters enclosed by apostrophes (') or quotation marks (").

character expression
A character constant, variable, function value, or another constant expression, separated by a concatenation operator (//); for example, DAY// ' FIRST'.

character set
A mapping of characters to their identifying numeric values. *See also* multibyte character set.

character storage unit
The unit of storage for holding a scalar value of default character type (and character length one) that is not a pointer. One character storage unit corresponds to one byte of memory.

character string
A sequence of contiguous characters; a character data value. *See also* character constant.

character substring
One or more contiguous characters in a character string.

child process
A process (child) initiated by another process (the parent). The child process can operate independently from the parent process. Further, the parent process can suspend or terminate without affecting the child process.

comment
Text that documents or explains a program. In free source form, a comment begins with an exclamation point (!), unless it appears in a Hollerith or character constant.

In fixed and tab source form, a comment begins with a letter C or an asterisk (*) in column 1. A comment can also begin with an exclamation point anywhere in a source line (except in a Hollerith or character constant) or in column 6 of a fixed-format line. The comment extends from the exclamation point to the end of the line.
The compiler does not process comments, but shows them in program listings. *See also* compiler directive.

common block
A physical storage area shared by one or more program units. This storage area is defined by a COMMON statement. If the common block is given a name, it is a named common block; if it is not given a name, it is a blank common.

compilation unit
The source file or files that are compiled together to form a single object file, possibly using interprocedural optimization across source files. Only one f90 command is used for each compilation, but one f90 command can specify that multiple compilation units be used.

compiler directive
A structured comment that tells the compiler to perform certain tasks when it compiles a source program unit. Compiler directives are usually compiler-specific. (Some Fortran compilers call these directives "metacommands".)

complex constant
A constant that is a pair of real or integer constants representing a complex number; the pair is separated by a comma and enclosed in parentheses. The first constant represents the real part of the number; the second constant represents the imaginary part. In DIGITAL Fortran, there are two types of complex constants: COMPLEX (COMPLEX(KIND=4)) and DOUBLE COMPLEX (COMPLEX(KIND=8)).

complex type
A data type that represents the values of complex numbers. The value is expressed as a complex constant. *See also* data type.

component
A part of a derived-type definition. There must be at least one component (intrinsic or derived type) in every derived-type definition.

concatenate
The combination of two items into one by placing one of the items after the other. In Fortran 90, the concatenation operator (//) is used to combine character items. *See also* character expression.

conformable
Pertains to dimensionality. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.

console
An interface that provides input and output to character-mode applications.

constant
A data object whose value does not change during the execution of a program; the value is defined at the time of compilation. A constant can be named (using the PARAMETER attribute or statement) or unnamed. An unnamed constant is called a literal constant. The value of a constant can be numeric or logical, or it can be a character string. *Contrast with* variable.

constant expression
An expression whose value does not change during program execution.

construct
A block of statements, beginning with CASE, DO, IF, FORALL, or WHERE statement, and ending with the appropriate termination statement.

contiguous

Pertaining to entities that are adjacent (next to one another) without intervening blanks (spaces); for example, contiguous characters or contiguous areas of storage.

control character
A character string, usually with an ASCII value between 0 and 31, used to communicate with devices such as printers, modems, and the like.

control edit descriptor
A format descriptor that directly displays text or affects the conversions performed by subsequent data edit descriptors. Except for the slash descriptor, control edit descriptors are nonrepeatable.

control statement
A statement that alters the normal order of execution by transferring control to another part of a program unit or a subprogram. A control statement can be conditional (such as the IF construct or computed GO TO statement) or unconditional (such as the STOP or GO TO statement).

critical section
An object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object.

D

data abstraction
A style of programming in which you define types to represent objects in your program, define a set of operations for objects of each type, and restrict the operations to only this set, making the types abstract. The Fortran 90 modules, derived types, and defined operators, support this programming paradigm.

data edit descriptor
A repeatable format descriptor that causes the transfer or conversion of data to or from its internal representation. In FORTRAN-77, this term was called a field descriptor.

data entity
A data object that has a data type. It is the result of the evaluation of an expression, or the result of the execution of a function reference (the function result).

data item
A unit of data (or value) to be processed. Includes constants, variables, arrays, character substrings, or records.

data object
A constant, variable, or part (subobject) of a constant or variable. Its type may be specified implicitly or explicitly.

data type
The properties and internal representation that characterize data and functions. Each intrinsic and user-defined data type has a name, a set of operators, a set of values, and a way to show these values in a program. The basic intrinsic data types are integer, real, complex, logical, and character. The data value of an intrinsic data type depends on the value of the type parameter. *See also* type parameter.

data type length specifier
The form *n appended to DIGITAL Fortran-specific data type names. For example, in REAL*4, the *4 is the data type length specifier.

deadlock
A bug where the execution of thread A is blocked indefinitely waiting for thread B to perform some action, while thread B is blocked waiting for thread A. For example, two threads on opposite ends of a named pipe can become deadlocked if each thread waits to read data written

by the other thread. A single thread can also deadlock itself. *See also* thread.

declaration

A statement or series of statements which specify attributes and properties of named entities, such as specifying the data type of named data objects. Declaration is a synonym for specification.

decorated name

An internal representation of a procedure name or variable name that contains information about where it is declared; for procedures, the information includes how it is called. Decorated names are mainly of interest in mixed-language programming, when calling Fortran routines from other languages.

default character

The kind type for character constants if no kind type parameter is specified. Currently, the only kind type parameter for character constants is CHARACTER(KIND=1), the default character kind.

default complex

The kind type for complex constants if no kind type parameter is specified. The default complex kind is affected by the compiler option specifying real size. If no compiler option is specified, default complex is COMPLEX(KIND=8) (COMPLEX*8). *See also* default real.

default integer

The kind type for integer constants if no kind type parameter is specified. The default integer kind is affected by compiler options specifying integer size. If no compiler option is specified, default integer is INTEGER(KIND=4) (INTEGER*4).

If a command line option affecting integer size has been specified, the integer has the kind specified, unless it is outside the range of the kind specified by the option. In this case, the kind type of the integer is the smallest integer kind which can hold the integer.

default logical

The kind type for logical constants if no kind type parameter is specified. The default logical kind is affected by compiler options specifying integer size. If no compiler option is specified, default logical is LOGICAL(KIND=4) (LOGICAL*4).*See also* default integer.

default real

The kind type for real constants if no kind type parameter is specified. The default real kind is affected by the compiler option specifying real size. If no compiler option is specified, default real is REAL(KIND=4) (REAL*4).

If a real constant is encountered that is outside the range for the default, an error occurs.

deferred-shape array

An array pointer (an array with the POINTER attribute) or an allocatable array (an array with the ALLOCATABLE attribute). The size in each dimension is determined by pointer assignment or when the array is allocated.

The declared bounds are specified by a colon (**:**).

definable

A property of variables. A variable is definable if its value can be changed by the appearance of its name or designator on the left of an assignment statement. An example of a variable that is not definable is an allocatable array that has not been allocated.

define

(1) To give a value to a data object during program execution. (2) To declare derived types and procedures.

defined assignment

An assignment statement that is not intrinsic, but is defined by a subroutine and an interface block. *See also* derived type.

defined operation
An operation that is not intrinsic, but is defined by a function subprogram containing a generic interface block with the specifier OPERATOR. *See also* interface block.

denormalized number
A computational floating-point result smaller than the lowest value in the normal range of a data type (the smallest representable normalized number). You cannot write a constant for a denormalized number.

derived type
A data type that is user-defined and not intrinsic. It requires a type definition to name the type and specify its components (which can be intrinsic or user-defined types). A structure constructor can be used to specify a value of derived type. A component of a structure is referenced using a percent sign (%).
Operations on objects of derived types (structures) must be defined by a function with an OPERATOR interface. Assignment for derived types can be defined intrinsically, or be redefined by a subroutine with an ASSIGNMENT interface. Structures can be used as procedure arguments and function results, and can appear in input and output lists. Also called a user-defined type. *See also* record , the first definition.

designator
A name that references a subobject (part of an object). A designator is the name of the object followed by a selector that selects the subobject. For example, B(3) is a designator for an array element. Also called a subobject designator. *See also* selector and subobject.

dimension
A range of values for one subscript or index of an array. An array can have from 1 to 7 dimensions. The number of dimensions is the rank of the array.

dimension bounds
See bounds.

direct access
A method for retrieving or storing data in which the data (record) is identified by the record number, or the position of the record in the file. The record is accessed directly (nonsequentially); therefore, all information is equally accessible. Also called random access. *Contrast with* sequential access.

DLL
See Dynamic Link Library.

double-byte character set (DBCS)
A mapping of characters to their identifying numeric values, in which each value is 2 bytes wide. Double-byte character sets are sometimes used for languages that have more than 256 characters. *See also* multibyte Character Set.

double-precision constant
A processor approximation to the value of a real number that occupies 8 bytes of memory and can assume a positive, negative, or zero value. The precision is greater than a constant of real (single-precision) type. For the precise ranges of the double-precision constants, see Data Representation in the *Programmer's Guide*. *See also* denormalized number.

driver program
On Windows NT, Windows 95, and DIGITAL UNIX systems, a program that is the user interface to the language compiler. It accepts command line options and file names and causes one or more language utilities or system programs to process each file.

dummy aliasing
The sharing of memory locations between dummy (formal) arguments and other dummy arguments or COMMON variables that are assigned.

dummy argument
A variable whose name appears in the parenthesized list following the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. A dummy argument takes the value of the corresponding actual argument in the calling program unit (through argument association). Also called a formal argument.

dummy array
A dummy argument that is an array.

dummy pointer
A dummy argument that is a pointer.

dummy procedure
Is a dummy argument that is specified as a procedure or appears in a procedure reference. The corresponding actual argument must be a procedure.

Dynamic Link Library (DLL)
A separate source module compiled and linked independently of the applications that use it. Applications access the DLL through procedure calls. The code for a DLL is not included in the user's executable image, but the compiler automatically modifies the executable image to point to DLL procedures at run time.

E

edit descriptor
A descriptor in a format specification. It can be a data edit descriptor, control edit descriptor, or string edit descriptor. *See also* control edit descriptor, data edit descriptor, and string edit descriptor.

element
See array element.

elemental
Pertains to an intrinsic operation, intrinsic procedure, or assignment statement that is independently applied to either of the following:
- The elements of an array
- Corresponding elements of a set of conformable arrays and scalars

end-of-file
The condition that exists when all records in a file open for sequential access have been read.

entity
A general term referring to any Fortran 90 concept; for example, a constant, a variable, a program unit, a statement label, a common block, a construct, an I/O unit and so forth.

environment variable
A symbolic variable that represents some element of the operating system, such as a path, a filename, or other literal data.

error number
An integer value denoting an I/O error condition, obtained by using the IOSTAT keyword in an I/O statement.

escape character
The character whose ascii value is 27, usually part of a string used to communicate commands to devices such as printers. *See also* control character.

exceptional values
For floating-point numbers, values outside the range of normalized numbers, including denormal (subnormal) numbers, infinity, Not-a-Number (NaN) values, zero, and other architecture-defined numbers.

executable construct
A CASE, DO, IF, WHERE, or FORALL construct.

executable program
A set of program units that include only one main program.

executable statement
A statement that specifies an action to be performed or controls one or more computational instructions.

explicit interface
A procedure interface whose properties are known within the scope of the calling program, and do not have to be assumed. These properties are the names of the procedure and its dummy arguments, the attributes of a procedure (if it is a function), and the attributes and order of the dummy arguments.
The following have explicit interfaces:
- Internal and module procedures (explicit by definition)
- Intrinsic procedures
- External procedures that have an interface block
- External procedures that are defined by the scoping unit and are recursive
- Dummy procedures that have an interface block

explicit-shape array
An array whose rank and bounds are specified when the array is declared.

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

extent
The size of (number of elements in) one dimension of an array.

external file
A sequence of records that exists in a medium external to the executing program.

external procedure
A procedure that is contained in an external subprogram. External procedures can be used to share information (such as source files, common blocks, and public data in modules) and can be used independently of other procedures and program units. Also called an external routine.

external subprogram
A subroutine or function that is not contained in a main program, module, or another subprogram. A module is not a subprogram.

F

field
Can be either of the following:
- A set of contiguous characters, considered as a single item, in a record or line.
- A substructure of a STRUCTURE declaration.

field descriptor
See data edit descriptor.

field separator
The comma (,) or slash (/) that separates edit descriptors in a format specification.

field width
The total number of characters in the field. *See also* field, the first definition.

file
A collection of logically related records. If the file is in internal storage, it is an internal file; if

the file is on an input/output device, it is an external file.

file access
The way records are accessed (and stored) in a file. The Fortran 90 file access modes are sequential and direct. On OpenVMS systems, you can also use a keyed mode of access.

file organization
The way records in a file are physically arranged on a storage device. Fortran 90 files can have sequential or relative organization. On OpenVMS systems, files can also have indexed organization.

fixed-length record type
A file format in which all the records are the same length.

focus window
Window to which keyboard input is directed.

foreground process
On DIGITAL UNIX systems, a process for which the command interpreter is waiting. Its process group is the same as that of its controlling terminal, so the process is allowed to read from or write to the terminal. *Contrast with* background process.

foreground window
The window with which the user is currently working. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

foreign file
An unformatted file that contains data from a foreign platform, such as data from a CRAY, IBM, or big endian IEEE machine.

format
A specific arrangement of data. A FORMAT statement specifies how data is to be read or written.

format specification
The part of a FORMAT statement that specifies explicit data arrangement. It is a list within parentheses that can include edit descriptors and field separators. A character expression can also specify format; the expression must evaluate to a valid format specification.

formatted data
Data written to a file by using formatted I/O statements. Such data contains ASCII representations of binary values.

formatted I/O statement
An I/O statement specifying a format for data transfer. The format specified can be explicit (specified in a format specification) or implicit (specified using list-directed or namelist formatting). *Contrast with* unformatted I/O statement. *See also* list-directed I/O statement and namelist I/O statement.

function
A series of statements that perform some operation and return a single value (through the function or result name) to the calling program unit. A function is invoked by a function reference in a main program unit or a subprogram unit.
In Fortran 90, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol. The function is invoked by the appearance of the new or extended operator in the expression (along with the appropriate operands). For example, the symbol * can be defined for logical operands, extending its intrinsic definition for numeric operands. *See also* function subprogram, statement function, and subroutine.

function reference
Used in an expression to invoke a function, it consists of the function name and its actual arguments. A function reference returns a value (through the function or result name) which is

used to evaluate the calling expression.

function result
The result value associated with a particular execution or call to a function. This result can be of any data type (including derived type) and can be array-valued. In a FUNCTION statement, the RESULT option can be used to give the result a name different from the function name. This option is required for a recursive function that directly calls itself.

function subprogram
A sequence of statements beginning with a FUNCTION (or optional OPTIONS) statement that is not in an interface block and ending with the corresponding END statement. *See also* function.

G

generic identifier
AA generic name, operator, or assignment specified in an INTERFACE statement that is associated with all of the procedures within the interface block. Also called a generic specification.

global entity
An entity (a program unit, common block, or external procedure) that can be used with the same meaning throughout the executable program. A global entity has global scope; it is accessible throughout an executable program. *See also* local entity.

global section
A data structure (for example, global COMMON) or shareable image section potentially available to all processes in the system.

H

handle
A 32-bit quantity which is an index into a table specific to a process. Handles have associated access control lists that the operating system uses to check against the security credentials of the process.

hexadecimal constant
A constant that is a string of hexadecimal (base 16) digits (range 0 to 9, or an uppercase or lowercase letter in the range A to F) enclosed by apostrophes or quotation marks and preceded by the letter Z.

High Performance Fortran
An extended version of Fortran 90 with features supporting parallel processing. DIGITAL Fortran 90 supports full High Performance Fortran (HPF), and compiles HPF programs for parallel execution.

Hollerith constant
A constant that is a string of printable ASCII characters preceded by *n*H, where *n* is the number of characters in the string (including blanks and tabs).

host
Either the main program or subprogram that contains an internal procedure, or the module that contains a module procedure. The data environment of the host is available to the (internal or module) procedure.

host association
The process by which a module procedure, internal procedure, or derived-type definition

accesses the entities of its host.

I

implicit interface
A procedure interface whose properties (the collection of names, attributes, and arguments of the procedure) are not known within the scope of the calling program, and have to be assumed. The information is assumed by the calling program from the properties of the procedure name and actual arguments in the procedure call.

implicit typing
The mechanism by which the data type for a variable is determined by the beginning letter of the variable name.

import library
A .LIB file that contains information about one or more dynamic-link libraries (DLLs), but does not contain the DLL's executable code. The linker uses an import library when building an executable module of a process, to provide the information needed to resolve the external references to DLL functions.

index
Can be any of the following:
- The variable used as a loop counter in a DO statement.
- An intrinsic function specifying the starting position of a substring inside a string.
- On OpenVMS systems, an internal data structure that provides a guide, based on key values, to file components in an indexed file.

indexed file organization
On OpenVMS systems, a file organization that allows random retrieval of records by key value and sequential retrieval of records within the key of reference. Each file contains records and a primary key index; it can also optionally have one or more alternate key indexes.

initialize
The assignment of an initial value to a variable.

initialization expression
A form of constant expression that is used to specify an initial value for an entity.

inlining
An optimization that replaces a subprogram reference (CALL statement or function invocation) with the replicated code of the subprogram.

input/output (I/O)
The data that a program reads or writes. Also, devices to read and write data.

inquiry function
An intrinsic function whose result depends on properties of the principal argument, not the value of the argument.

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

intent
An attribute of a dummy argument that is not a procedure or a pointer. It indicates whether the argument is used to transfer data into the procedure, out of the procedure, or both.

interactive process
A process that must periodically get user input to do its work. *Contrast with* background process or batch process.

interface

The properties of a procedure, consisting of: specifications of the attributes for a function result, the specification of dummy argument attributes, and the information in the procedure heading.

interface block
The sequence of statements starting with an INTERFACE statement and ending with the corresponding END INTERFACE statement.

interface body
The sequence of statements in an interface block starting with a FUNCTION or SUBROUTINE statement and ending with the corresponding END statement. Also called a procedure interface body.

internal file
The designated internal storage space (or variable buffer) that is manipulated during input and output. An internal file can be a character variable, character array, character array element, or character substring. In general, an internal file contains one record. However, an internal file that is a character array has one record for each array element.

internal procedure
A procedure (other than a statement function) that is contained within an internal subprogram. The program unit containing an internal procedure is called the host of the internal procedure. The internal procedure (which appears between a CONTAINS and END statement) is local to its host and inherits the host's environment through host association.

internal subprogram
A subprogram contained in a main program or another subprogram.

intrinsic
Describes entities defined by the Fortran 90 language (such as data types and procedures). Intrinsic entities can be used freely in any scoping unit.

intrinsic procedure
A subprogram supplied as part of the Fortran 90 library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available to any Fortran 90 program unit (unless specifically overridden by an EXTERNAL statement or a procedure interface block). Also called a built-in or library procedure.

invoke
To call upon; used especially with reference to subprograms. For example, to invoke a function is to execute the function.

iteration count
The number of executions of the DO range, which is determined as follows:

```
[(terminal value - initial value + increment value) / increment value]
```

K

key
On OpenVMS systems, a value in a file of indexed organization that the system uses to build indexes into the file. Each key is identified by its location within the component, its length, and its data type. Also called the key field. *See also* alternate key, index, and primary key.

keyed access
On OpenVMS systems, a method for retrieving or writing data in which the data (a record) is identified by specifying the information in a key field of the record. *See also* key.

key of reference

On OpenVMS systems, a key used to determine the index to use when sequentially accessing components of an indexed file. *See also* key, indexed file organization, and sequential access.

keyword
(1) Part of the syntax of a statement (syntax keyword). These keywords are not reserved. (2) A dummy argument name.

kind type parameter
Indicates the range of an intrinsic data type. For real and complex types, it also indicates precision. If a specific kind type parameter is not specified (for example, INTEGER), the kind type is the default for that type (for example, default integer). *See also* default character, default complex, default integer, default logical, and default real.

L

label
An integer, from 1 to 5 digits long, that is used to identify a statement. For example, labels can be used to refer to a FORMAT statement or branch target statement.

language extension
A DIGITAL Fortran language element or interpretation that is not part of the Fortran 90 standard.

lexical token
A sequence of one or more characters that have an indivisible interpretation. A lexical token is the smallest meaningful unit (a basic language element) of a Fortran 90 statement; for example, constants, and statement keywords.

line
A source form record consisting of 0 or more characters. A standard Fortran 90 line is limited to a maximum of 132 characters.

linker
A system program that creates an executable program from one or more object files produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create OpenVMS executable images or DIGITAL UNIX, Windows NT, and Windows 95 executable files.

list-directed I/O statement
An implicit, formatted I/O statement that uses an asterisk (*) specifier rather than an explicit format specification. *See also* formatted I/O statement and namelist I/O statement.

listing
A printed copy of a program.

literal constant
A constant without a name. In Fortran 77, this was called simply a constant.

little endian
A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the lowest addressed byte. This is the method used on DIGITAL systems. *Contrast with* big endian.

local entity
An entity that can be used only within the context of a subprogram (its scoping unit); for example, a statement label. A local entity has local scope. *See also* global entity.

local optimization
Refers to enabling local optimizations within the source program unit, recognition of common expressions, and integer multiplication and division expansion (using shifts). The order of compilation of procedures is determined from the call graph. *See also* optimization.

local symbol
A name defined in a program unit that is not accessible outside of that program unit.

logical constant
A constant that specifies the value .TRUE. or .FALSE..

logical expression
An integer or logical constant, variable, function value, or another constant expression, joined by a relational or logical operator. The logical expression is evaluated to a value of either true or false. For example, .NOT. 6.5 + (B .GT. D).

logical operator
A symbol that represents an operation on logical expressions. The logical operators are .AND., .OR., .NEQV., .XOR., .EQV., and .NOT..

logical unit
A channel in memory through which data transfer occurs between the program and the device or file. *See also* unit identifier.

longword
Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered 0 to 31. The address of the longword is the address of the byte containing bit 0. When the longword is interpreted as a signed integer, bit 31 is the sign bit. The value of signed integers is in the range -2**31 to 2**31-1. The value of unsigned integers is in the range 0 to 2**32-1.

loop
A group of statements that are executed repeatedly until an ending condition is reached.

M

main program
A program unit containing a PROGRAM statement (or not containing a SUBROUTINE, FUNCTION, or BLOCK DATA statement). The main program is the first program unit to receive control when a program is run, and exercises control over subprograms. *Contrast with* subprogram.

makefile
On DIGITAL UNIX systems, an argument to the make command containing a sequence of entries that specify dependences. On Windows NT and Windows 95 systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependences. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs. For more information, see Building Projects with NMAKE on Windows NT and Windows 95 systems, or make(1) on DIGITAL UNIX systems.

many-one array section
An array section with a vector subscript having two or more elements with the same value.

metacommand
See compiler directive.

misaligned data
Data not aligned on a natural boundary. *See also* natural boundary.

module
A program unit that contains specifications and definitions that other program units can access (unless the module entities are declared PRIVATE). Modules are referenced in USE statements.

module procedure
A subroutine or function defined within a module subprogram (the module procedure's host). The module procedure appears between a CONTAINS and END statement in its host module,

and inherits the host module's environment through host association. A module procedure can be declared PRIVATE to the module; it is public by default.

module subprogram
A subprogram that is contained in a module. (It cannot be an internal subprogram.)

multibyte character set
A character set in which each character is identified by using more than one byte. Although Unicode characters are 2 bytes wide, the Unicode character set is not referred to by this term.

N

name
Identifies an entity within a Fortran program unit (such as a variable, function result, common block, named constant, procedure, program unit, namelist group, or dummy argument). In FORTRAN 77, this term was called a symbolic name.

name association
Pertains to argument, host, or use association.

named common block
A common block (one or more contiguous areas of storage) with a name. Common blocks are defined by a COMMON statement.

named constant
A constant that has a name. In FORTRAN 77, this term was called a symbolic constant.

namelist I/O statement
An implicit, formatted I/O statement that uses a namelist group specifier rather than an explicit format specifier. *See also* formatted I/O statement and list-directed I/O statement.

natural boundary
The virtual address of a data item that is the multiple of the size of its data type. For example, a REAL(KIND=8) (REAL*8) data item aligned on natural boundaries has an address that is a multiple of eight.

naturally aligned record
A record that is aligned on a hardware-specific natural boundary; each field is naturally aligned. (For more information, see Data Alignment Considerations in the *Programmer's Guide*.) *Contrast with* packed record.

nesting
The placing of one entity (such as a construct, subprogram, format specification, or loop) inside another entity of the same kind. For example, nesting a loop within another loop (a nested loop), or nesting a subroutine within another subroutine (a nested subroutine).

nonexecutable statement
A Fortran 90 statement that describes program attributes, but does not cause any action to be taken when the program is executed.

nonsignaled
The state of an object used for synchronization in one of the wait functions is either signaled or nonsignaled. A nonsignaled state can prevent the wait function from returning. See also wait function.

numeric expression
A numeric constant, variable, or function value, or combination of these, joined by numeric operators and parentheses, so that the entire expression can be evaluated to produce a single numeric value. For example, -L or X+(Y-4.5*Z).

numeric operator
A symbol designating an arithmetic operation. In Fortran 90, the symbols +, -, *, /, and ** are

used to designate addition, subtraction, multiplication, division, and exponentiation, respectively.

numeric storage unit
The unit of storage for holding a non-pointer scalar value of type default real, default integer, or default logical. One numeric storage unit corresponds to 4 bytes of memory.

numeric type
Integer, real, or complex type.

O

object
(1) An internal structure that represents a system resource such as a file, a thread, or a graphic image. (2) A data object.

object file
The binary output of a language processor (such as an assembler or compiler), which can either be executed or used as input to the linker.

obsolescent feature
A feature of FORTRAN 77 that is considered to be redundant in Fortran 90. These features are still in frequent use.

octal constant
A constant that is a string of octal (base 8) digits (range of 0 to 7) enclosed by apostrophes or quotation marks and preceded by the letter O.

operand
The passive element in an expression on which an operation is performed. Every expression must have at least one operand. For example, in I .NE. J, I and J are operands. *Contrast with* operator.

operation
A computation involving one or two operands.

operator
The active element in an expression that performs an operation. An expression can have zero or more operators. Intrinsic operators are arithmetic (+, -, *, /, and **) or logical (.AND., .NOT., and so on). For example, in I .NE. J, .NE. is the operator.
Executable programs can define operators which are not intrinsic.

optimization
The process of producing efficient object or executing code that takes advantage of the hardware architecture to produce more efficient execution.

optional argument
A dummy argument that has the OPTIONAL attribute (or is included in an OPTIONAL statement in the procedure definition). Such an argument does not have to be associated with an actual argument.

order of subscript progression
A characteristic of a multidimensional array in which the leftmost subscripts vary most rapidly.

overflow
An error condition occurring when an arithmetic operation yields a result that is larger than the maximum value in the range of a data type.

P

packed record
A record that starts on an arbitrary byte boundary; each field starts in the next unused byte. *Contrast with* naturally aligned record.

pad
The filling of unused positions in a field or character string with dummy data (such as zeros or blanks).

parameter
Can be either of the following:
- In general, any quantity of interest in a given situation; often used in place of the term "argument".
- A Fortran 90 named constant.

parent window
A window that has one or more child windows.

pathname
On Windows NT, Windows 95, and DIGITAL UNIX systems, the path from the root directory to a subdirectory or file. *See also* root.

pipe
A connection that allows one program to get its input directly from the output of another program

platform
A combination of operating system and hardware that provides a distinct environment in which to use a software product (for example, Microsoft Windows 95 on Intel processors).

pointer
Is one of the following:
- A Fortran 90 pointer
  A data object that has the POINTER attribute. To be referenced or defined, it must be "pointer-associated" with a target (have storage space associated with it). If the pointer is an array, it must be pointer-associated to have a shape. *See also* pointer association.
- A DIGITAL Fortran 77 pointer
  A data object that contains the address of its paired variable. This is also called an integer pointer or a Cray® pointer.

pointer assignment
The association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.

pointer association
The association of storage space to a Fortran 90 pointer by means of a target. A pointer is associated with a target after pointer assignment or the valid execution of an ALLOCATE statement.

precision
The number of significant digits in a real number. *See also* double-precision constant, kind type parameter, and single-precision constant.

primary
The simplest form of an expression. A primary can be any of the following data objects:
- A constant
- A constant subobject (parent is a constant)
- A variable (scalar, structure, array, or pointer; an array cannot be assumed size)
- An array constructor

- A structure constructor
- A function reference
- An expression in parentheses

primary key
On OpenVMS systems, the required key within the data records of an indexed file. This key is used to determine the placement of records within the file and to build the primary index.

primary thread
The initial thread of a process. Also called the main thread or thread 1.

procedure
A computation that can be invoked during program execution. It can be a subroutine or function, an internal, external, dummy or module procedure, or a statement function. A subprogram can define more than one procedure if it contains an ENTRY statement. *See also* subprogram.

procedure interface
The statements that specify the name and characteristics of a procedure, the name and attributes of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. If these properties are all known to the calling program, the procedure interface is explicit; otherwise it is implicit.

process object
A virtual address space, security profile, a set of threads that execute in the address space of the process, and a set of resources visible to all threads executing in the process. Several thread objects can be associated with a single process.

program unit
The fundamental component of an executable program. A sequence of statements and comment lines. It can be a main program, a module, an external subprogram, or a block data program unit.

Q

quadword
Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered 0 to 63. (Bit 63 is used as the sign bit.) A quadword is identified by the address of the word containing the low-order bit (bit 0). The value of a signed quadword integer is in the range -2**63 to 2**63-1.

R

random access
See direct access.

rank
The number of dimensions of an array. A scalar has a rank of zero.

rank-one object
A data structure comprising scalar elements with the same data type and organized as a simple linear sequence. *See also* scalar.

real constant
A constant that is a number written with a decimal point, exponent, or both. It can have single precision (REAL(4)) or double precision (REAL(8)). On OpenVMS and DIGITAL UNIX systems, it can also have quad precision (REAL(16)).

record
Can be either of the following:
- A set of logically related data items (in a file) that is treated as a unit; such a record contains one or more fields. This definition applies to I/O records and items that are declared in a record structure.
- One or more data items that are grouped in a structure declaration and specified in a RECORD statement.

record access
The method used to store and retrieve records in a file.

record structure declaration
A block of statements that define the fields in a record. The block begins with a STRUCTURE statement and ends with END STRUCTURE. The name of the structure must be specified in a RECORD statement.

record type
The property that determines whether records in a file are all the same length, of varying length, or use other conventions to define where one record ends and another begins.

recursion
Pertains to a subroutine or function that directly or indirectly references itself.

reference
Can be any of the following:
- For a data object, the appearance of its name, designator, or associated pointer where the value of the object is required. When an object is referenced, it must be defined.
- For a procedure, the appearance of its name, operator symbol, or assignment symbol that causes the procedure to be executed. Procedure reference is also called "calling" or "invoking" a procedure.
- For a module, the appearance of its name in a USE statement.

relational expression
An expression containing one relational operator and two operands of numeric or character type. The result is a value that is true or false. For example, A-C .GE. B+2 or DAY .EQ. ’MONDAY’.

relational operator
The symbols used to express a relational condition or expression. The relational operators are (.EQ., .NE., .LT., .LE., .GT., and .GE.).

relative file organization
A file organization that consists of a series of component positions, called cells, numbered consecutively from 1 to n. DIGITAL Fortran 90 uses these numbered, fixed-length cells to calculate the component’s physical position in the file.

relative pathname
A directory path expressed in relation to any directory other than the root directory. *Contrast with* absolute pathname.

root
On Windows NT and Windows 95 systems, the top-level directory on a disk drive; it is represented by a backslash (\). For example, C:\ is the root directory for drive C.
On DIGITAL UNIX systems, the top-level directory in the file system; it is represented by a slash (/).

run time
The time during which a computer executes the statements of a program.

S

saved object
A variable that retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement in the scoping unit containing the declaration.

scalar
Pertaining to data items with a rank of zero. A single data object of any intrinsic or derived data type. *Contrast with* array. *See also* rank-one object.

scalar memory reference
A reference to a scalar variable, scalar record field, or array element that resolves into a single data item (having a data type) and can be assigned a value with an assignment statement. It is similar to a scalar reference, but it excludes constants, character substrings, and expressions.

scalar reference
A reference to a scalar variable, scalar record field, derived-type component, array element, constant, character substring, or expression that resolves into a single data item having a data type.

scalar variable
A variable name specifying one storage location.

scale factor
A number indicating the location of the decimal point in a real number and, if there is no exponent, the size of the number on input.

scope
The portion of a program in which a declaration or a particular name has meaning. Scope can be global (throughout an executable program), scoping unit (local to the scoping unit), or statement (within a statement, or part of a statement).

scoping unit
The part of the program in which a name has meaning. It is one of the following:

- A program unit or subprogram
- A derived-type definition
- A procedure interface body

Scoping units can not overlap, though one scoping unit can contain another scoping unit. The outer scoping unit is called the host scoping unit.

screen coordinates
Coordinates relative to the upper left corner of the screen.

section subscript
A subscript list (enclosed in parentheses and appended to the array name) indicating a portion (section) of an array. At least one of the subscripts in the list must be a subscript triplet or vector subscript. The number of section subscripts is the rank of the array. *See also* array section, subscript, subscript triplet, and vector subscript.

seed
A value (which can be assigned to a variable) that is required in order to properly determine the result of a calculation; for example, the argument i in the random number generator (RAN) function syntax:

```
y = RAN (i).
```

selector
A mechanism for designating the following:

- Part of a data object (an array element or section, a substring, a derived type, or a structure component)
- The set of values for which a CASE block is executed

sequence
A set ordered by a one-to-one correspondence with the numbers 1 through n, where *n* is the total number of elements in the sequence. A sequence can be empty (contain no elements).

sequential access
A method for retrieving or storing data in which the data (record) is read from, written to, or removed from a file based on the logical order (sequence) of the record in the file. (The record cannot be accessed directly.) *Contrast with* direct access.

sequential file organization
A file organization in which records are stored one after the other, in the order in which they were written to the file.

shape
The rank and extents of an array. Shape can be represented by a rank-one array (vector) whose elements are the extents in each dimension.

shape conformance
Pertains to the rule concerning operands of binary intrinsic operations in expressions**:** to be in shape conformance, the two operands must both be arrays of the same shape, or one or both of the operands must be scalars.

short field termination
The use of a comma (,) to terminate the field of a numeric data edit descriptor. This technique overrides the field width (w) specification in the data edit descriptor and therefore avoids padding of the input field. The comma can only terminate fields less than w characters long. *See also* data edit descriptor.

signal
The software mechanism used to indicate that an exception condition (abnormal event) has been detected. For example, a signal can be generated by a program or hardware error, or by request of another program.

single-precision constant
A processor approximation of the value of a real number that occupies 4 bytes of memory and can assume a positive, negative, or zero value. The precision is less than a constant of double-precision type. For the precise ranges of the single-precision constants, see Data Representation in the *Programmer's Guide*. *See also* denormalized number.

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

source file
A program or portion of a program library, such as an object file, or image file.

specification expression
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.

specification statement
A nonexecutable statement that provides information about the data used in the source program. Such a statement can be used to allocate and initialize variables, arrays, records, and structures, and define other characteristics of names used in a program.

statement
An instruction in a programming language that represents a step in a sequence of actions or a set of declarations. In Fortran 90, an ampersand can be used to continue a statement from one line to another, and a semicolon can be used to separate several statements on one line. There are two main classes of statements: executable and nonexecutable.

statement entity
An entity identified by a lexical token whose scope is a single statement or part of a statement.

statement function
A function whose definition is contained in a single statement.

statement function definition
A statement that defines a statement function. Its form is the statement function name (followed by its optional dummy arguments in parentheses), followed by an equal sign (=), followed by a numeric, logical, or character expression.
A statement function definition must precede all executable statements and follow all specification statements.

statement keyword
A word that begins the syntax of a statement. All program statements (except assignment statements and statement function definitions) begin with a statement keyword. Examples are INTEGER, DO, IF, and WRITE.

statement label
See label.

static variable
A variable whose storage is allocated for the entire execution of a program.

storage association
The relationship between two storage sequences when the storage unit of one is the same as the storage unit of the other. Storage association is provided by the COMMON and EQUIVALENCE statements. For modules, pointers, allocatable arrays, and automatic data objects, the SEQUENCE statement defines a storage order for structures.

storage location
An addressable unit of main memory.

storage sequence
A sequence of any number of consecutive storage units. The size of a storage sequence is the number of storage units in the storage sequence. A sequence of storage sequences forms a composite storage sequence. *See also* storage association and storage unit.

storage unit
In a storage sequence, the number of storage units needed to represent one real, integer, logical, or character value. *See also* character storage unit, numeric storage unit, and storage sequence.

stride
The increment between subscript values, specified in a subscript triplet. If it is omitted, it is assumed to be one.

string edit descriptor
A format descriptor that transfers characters to an output record.

structure
Can be either of the following:
- A scalar data object of derived (user-defined) type.
- An aggregate entity containing one or more fields or components.

structure component
Can be either of the following:
- One of the components of a structure.
- An array whose elements are components of the elements of an array of derived type.

structure constructor
A mechanism that is used to specify a scalar value of a derived type. A structure constructor is the name of the type followed by a parenthesized list of values for the components of the type.

subobject
Part of a data object (parent object) that can be referenced and defined separately from other parts of the data object. A subobject can be an array element, an array section, a substring, a

derived type, or a structure component. Subobjects are referenced by designators and can be considered to be data objects themselves. *See also* designator.

subobject designator
See designator.

subprogram
A user-written function or subroutine subprogram that can be invoked from another program unit to perform a specific task. Note that in FORTRAN 77, a block data program unit was also called a subprogram.

subroutine
procedure that can return many values, a single value, or no value to the calling program unit (through arguments). A subroutine is invoked by a CALL statement in another program unit. In Fortran 90, a subroutine can also be used to define a new form of assignment (defined assignment), which is different from those intrinsic to Fortran 90. Such assignments are invoked with assignment syntax (using the = symbol) rather than the CALL statement. *See also* function, statement function, and subroutine subprogram.

subroutine subprogram
A sequence of statements starting with a SUBROUTINE (or optional OPTIONS) statement and ending with the corresponding END statement. *See also* subroutine.

subscript
A scalar integer expression (enclosed in parentheses and appended to the array name) indicating the position of an array element. The number of subscripts is the rank of the array. *See also* array element.

subscript triplet
An item in a section subscript list specifying a range of values for the array section. A subscript triplet contains at least one colon and has three optional parts: a lower bound, an upper bound, and a stride. *Contrast with* vector subscript. *See also* array section and section subscript.

substring
A contiguous portion of a scalar character string. Do not confuse this with the substring selector in an array section, where the result is another array section, not a substring.

symbolic name
See name.

syntax
The formal structure of a statement or command string.

T

target
The named data object associated with a pointer (in the form pointer-object => target). A target is declared in a type declaration statement that contains the TARGET attribute. *See also* pointer and pointer association.

thread
The smallest unit of execution for which the operating system allocates CPU time. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each process has at least one thread of execution.

transformational function
An intrinsic function that is not an elemental or inquiry function. A transformational function usually changes an array actual argument into a scalar result or another array, rather than applying the argument element by element.

truncation

Can be either of the following:

- A technique that approximates a numeric value by dropping its fractional value and using only the integer portion.
- The process of removing one or more characters from the left or right of a number or string.

type declaration statement

A nonexecutable statement specifying the data type of one or more variables: an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, LOGICAL, or TYPE statement. Also called a type declaration or type specification.

type parameter

Defines an intrinsic data type. The type parameters are kind and length. The kind type parameter (KIND=) specifies the range for the integer data type, the precision and range for real and complex data types, and the machine representation method for the character and logical data types. The length type parameter (LEN=) specifies the length of a character string.*See also* kind type parameter.

U

ultimate component

For a derived type or a structure, a component that is of intrinsic type or has the POINTER attribute, or an ultimate component of a component that is a derived type and does not have the POINTER attribute.

unary operator

An operator that operates on one operand. For example, the minus sign in -A and the .NOT. operator in .NOT. (J .GT. K).

undefined

For a data object, the property of not having a determinate value.

underflow

An error condition occurring when the result of an arithmetic operation yields a result that is smaller than the minimum value in the range of a data type. For example, in unsigned arithmetic, underflow occurs when a result is negative. *See also* denormalized number.

unformatted data

Data written to a file by using unformatted I/O statements; for example, binary numbers.

unformatted I/O statement

An I/O statement that does not contain format specifiers and therefore does not translate the data being transferred. *Contrast with* formatted I/O statement.

unformatted record

A record that is transmitted in internal format between internal and external storage.

unit identifier

The identifier that specifies an external unit or internal file. The identifier can be any one of the following:

- An integer expression whose value must be zero or positive
- An asterisk (*) that corresponds to the default (or implicit) I/O unit
- The name of a character scalar memory reference or character array name reference for an internal file

Also called a device code, or logical unit number.

unspecified storage unit

A unit of storage for holding a pointer or a scalar that is not a pointer and is of type other than default integer, default character, or default real.

use association
The process by which the entities in a module are made accessible to other scoping units (through a USE statement in the scoping unit).

user-defined type
See derived type.

V

variable
A data object (stored in a memory location) whose value can change during program execution. A variable can be a named data object, an array element, an array section, a structure component, or a substring. In FORTRAN 77, a variable was always scalar and named. *Contrast with* constant.

variable format expression
A numeric expression enclosed in angle brackets (<>) that can be used in a FORMAT statement. If necessary, it is converted to integer type before use.

variable-length record type
A file format in which records may be of different lengths.

vector subscript
A rank-one array of integer values used as a section subscript to select elements from a parent array. Unlike a subscript triplet, a vector subscript specifies values (within the declared bounds for the dimension) in an arbitrary order. *Contrast with* subscript triplet. *See also* array section and section subscript.

W

wait function
A function that blocks the execution of a calling thread until a specified set of conditions has been satisfied.