Compaq Visual Fortran
Language Reference

Date: August, 2000

Software Version: Visual Fortran Version 6.5


Compaq Computer Corporation
Houston, Texas
Compaq Computer Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Compaq or an authorized sublicensor.

Compaq shall not be liable for technical or editorial errors or omissions contained herein. The information in this document is subject to change without notice.

© 1997-2000 Compaq Computer Corporation

Compaq, the COMPAQ logo, DEC, DIGITAL, VAX, and VMS are registered in the U.S. Patent and Trademark Office.

DEC Fortran, OpenVMS, Tru64 UNIX, and VAX FORTRAN are trademarks of Compaq Information Technologies, L.P.


Intel and Pentium are trademarks of Intel Corporation.

CRAY is a registered trademark of Cray Research, Inc.

IBM is a registered trademark of International Business Machines, Inc.

IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

IMSL and Visual Numerics are registered trademarks of Visual Numerics, Inc.

Linux is a registered trademark of Linus Torvalds.

OpenGL is a registered trademark of Silicon Graphics, Inc.
OpenMP and the OpenMP logo are trademarks of OpenMP Architecture Review Board.

Sun Microsystems is a registered trademark and Java is a trademark of Sun Microsystems, Inc.

UNIX is a trademark of the Open Group.

All other product names mentioned herein may be trademarks of their respective companies.
Introduction to the Language Reference

This manual contains the description of the *Compaq Fortran* (formerly DIGITAL 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 95/90 language.

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

In this document, links are denoted by a 📚 or 📚 when you pass your pointer over a blue-colored term. In either case, click on the link to see further information.

This document contains the following major sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overview</strong></td>
<td>Describes language standards, language compatibility, and Fortran 95/90 features.</td>
</tr>
<tr>
<td><strong>Program Structure, Characters, and Source Forms</strong></td>
<td>Describes program structure, the Fortran 95/90 character set, and source forms.</td>
</tr>
<tr>
<td><strong>Data Types, Constants, and Variables</strong></td>
<td>Describes intrinsic and derived data types, constants, variables (scalars and arrays), and substrings.</td>
</tr>
<tr>
<td><strong>Expressions and Assignment Statements</strong></td>
<td>Describes expressions and assignment.</td>
</tr>
<tr>
<td><strong>Specification Statements</strong></td>
<td>Describes specification statements, which declare the attributes of data objects.</td>
</tr>
<tr>
<td><strong>Dynamic Allocation</strong></td>
<td>Describes dynamic allocation of data objects.</td>
</tr>
<tr>
<td><strong>Execution Control</strong></td>
<td>Describes constructs and statements that can transfer control within a program.</td>
</tr>
<tr>
<td>Topic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Program Units and Procedures</strong></td>
<td>Describes program units (including modules), subroutines and functions, and procedure interfaces.</td>
</tr>
<tr>
<td><strong>Intrinsic Procedures</strong></td>
<td>Contains general information on Visual Fortran intrinsic procedures. Each intrinsic is fully described in the A-Z Reference.</td>
</tr>
<tr>
<td><strong>Data Transfer I/O Statements</strong></td>
<td>Describes data transfer input/output (I/O) statements.</td>
</tr>
<tr>
<td><strong>I/O Formatting</strong></td>
<td>Describes the rules for I/O formatting.</td>
</tr>
<tr>
<td><strong>File Operation I/O Statements</strong></td>
<td>Describes auxiliary I/O statements you can use to perform file operations on Windows NT (including Windows 2000), Windows 95/98, Tru64 UNIX, and Linux systems.</td>
</tr>
<tr>
<td><strong>Compilation Control Statements</strong></td>
<td>Describes compilation control statements.</td>
</tr>
<tr>
<td><strong>Compiler Directives</strong></td>
<td>Describes compiler directives.</td>
</tr>
<tr>
<td><strong>Scope and Association</strong></td>
<td>Describes scope and association.</td>
</tr>
<tr>
<td><strong>Obsolescent and Deleted Language Features</strong></td>
<td>Describes obsolescent language features in Fortran 95 and Fortran 90.</td>
</tr>
<tr>
<td><strong>Additional Language Features</strong></td>
<td>Describes some statements and language features supported for programs written in older versions of Fortran.</td>
</tr>
<tr>
<td><strong>Character and Key Code Charts</strong></td>
<td>Describes the Visual Fortran character sets available on Windows NT (including Windows 2000) and Windows 95/98 systems.</td>
</tr>
<tr>
<td><strong>Data Representation Models</strong></td>
<td>Describes data representation models for numeric intrinsic functions.</td>
</tr>
<tr>
<td><strong>Run-Time Library Routines on Tru64 UNIX and Linux Systems</strong></td>
<td>Summarizes the library routines available for Tru64 UNIX and Linux systems.</td>
</tr>
<tr>
<td><strong>FORTRAN 77 Syntax</strong></td>
<td>Summarizes the syntax for features of the ANSI FORTRAN 77 Standard.</td>
</tr>
<tr>
<td><strong>Summary of Language Extensions</strong></td>
<td>Summarizes Compaq Fortran extensions to the Fortran 95 Standard.</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>A-Z Reference</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td>Contains abbreviated definitions of some commonly used terms in this manual.</td>
</tr>
</tbody>
</table>

**Other Sources of Information**

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


Compaq 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
- Syntax Conventions
- Platform Labels

### General Conventions

The *Language Reference* uses the following general conventions. (Note that in most cases, blanks are not significant in Fortran 95/90.)

<table>
<thead>
<tr>
<th>When you see this</th>
<th>Here is what it means</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extensions to Fortran 95</strong></td>
<td>This color indicates extensions to the Fortran 95 Standard. These extensions may or may not be implemented by other compilers that conform to the language standard.</td>
</tr>
<tr>
<td><strong>OUT.TXT, ANOVA.EXE, COPY, LINK, FL32</strong></td>
<td>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).</td>
</tr>
<tr>
<td>! Comment line WRITE (<em>,</em>) 'Hello &amp; World'</td>
<td>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 ( &amp; ) after the code at the end of a line to be continued and before the code on the following line.</td>
</tr>
<tr>
<td><strong>AUTOMATIC, INTRINSIC, WRITE</strong></td>
<td>Bold capital letters indicate Fortran 95/90 statements, functions, subroutines, and keywords. Keywords are a required part of statement syntax, unless enclosed in brackets as explained below. In the sentence, &quot;The following steps occur when a <strong>DO WHILE</strong> statement is executed,&quot; the phrase <strong>DO WHILE</strong> is a Fortran 95/90 keyword.</td>
</tr>
</tbody>
</table>
**other keywords**  | Bold lowercase letters are used for keywords of other languages. In the sentence, "A Fortran 95/90 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.

**compiler option**  | This term refers to Windows NT (including Windows 2000) and Windows 9* options, OpenVMS qualifiers, and Tru64 UNIX and Linux flags that can be used on the compiler command line.

**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.
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 colored 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® 98 on x86 is a platform.

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS</td>
<td>Applies to OpenVMS™ on Alpha processors.</td>
</tr>
<tr>
<td>U*X</td>
<td>Applies to Tru64 UNIX® and Linux® on Alpha processors.</td>
</tr>
<tr>
<td>TU*X</td>
<td>Applies to Tru64 UNIX® on Alpha processors.</td>
</tr>
<tr>
<td>WNT</td>
<td>Applies to Microsoft Windows NT® 4.0 on Alpha and x86 processors, and also Windows® 2000 on x86 processors.</td>
</tr>
<tr>
<td>W9*</td>
<td>Applies to Microsoft Windows 98 and Windows 95 on x86 processors.</td>
</tr>
<tr>
<td>Alpha</td>
<td>Applies to OpenVMS, Tru64 UNIX, Linux, and Microsoft Windows NT on Alpha processors. With Visual Fortran, only the Professional and Enterprise Editions support Alpha processors (see System Requirements and Optional Software in <em>Visual Fortran Installing and Getting Started</em>).</td>
</tr>
<tr>
<td>x86</td>
<td>Applies to Microsoft Windows NT 4.0, Windows 2000, Windows 98, and Windows 95 on x86 processors (see System Requirements and Optional Software in <em>Visual Fortran Installing and Getting Started</em>).</td>
</tr>
</tbody>
</table>
Overview

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

**Graphic Representation of Compaq Fortran**

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

For more information, see:
- Language Standards Conformance
- Language Compatibility
- Fortran 95 Features
- Fortran 90 Features

**Language Standards Conformance**

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

Compaq Fortran provides a number of extensions to the Fortran 95 Standard. In the language reference, extensions are displayed in this color.


1This is the same as International Standards Organization standard ISO/IEC 1539-1:1997 (E).
2This is the same as International Standards Organization standard ISO/IEC 1539:1991 (E).

Language Compatibility

Compaq Fortran is highly compatible with Compaq Fortran 77 (formerly DIGITAL Fortran 77) on supported systems, and it is substantially compatible with PDP-11 and VAX FORTRAN 77.

Fortran 95 Features

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

Some features of FORTRAN 77 have been deleted, but they are fully supported by Compaq Fortran. These features are listed in Deleted Language Features in Fortran 95.

Other features have been identified as obsolescent in Fortran 95. These features are listed in Obsolescent Language Features in Fortran 90.

Fortran 95 features are described in the following sections:

- New Features
- Improved Features
New Features

The following Fortran 95 features are new to 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.
  
  Compaq 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.
  
  Compaq 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.

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

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

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

- New obsolescent features

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

Compaq Fortran fully supports features deleted in Fortran 95.

**Improved Features**

The following Fortran 95 features improve previous Fortran features:

- Derived-type structure **default initialization**

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

- Pointer initialization

  In Fortran 90, there was no way to define the initial value of a pointer. You can now specify **default initialization** for a pointer. See also Derived-Type Definition.

- Automatic deallocation of allocatable arrays

  Allocatable arrays whose status is allocated upon routine exit are now automatically deallocated. For more information, see Deallocation of Allocatable Arrays.

  Compaq Fortran previously provided this feature.

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

- **Enhanced \texttt{SIGN} intrinsic function**

  When compiler option /assume:minus0 is specified, the \texttt{SIGN} function can now distinguish between positive and negative zero if the processor is capable of doing so.

- **Printing of -0.0**

  When 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 \texttt{WHERE} construct**

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

- **Generic identifier allowed in \texttt{END INTERFACE} statement**

  The \texttt{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. Compaq Fortran previously provided this feature as a language extension.

**Fortran 90 Features**

Fortran 90 offers significant enhancements to FORTRAN 77. Some of the features of Fortran 90 were implemented as language extensions in earlier versions of Compaq (formerly DIGITAL) Fortran.

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

Fortran 90 features are described in the following sections:

- 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, `PROGRAM` 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 also Obsolescent 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.
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 95/90, this type of program unit can be replaced by a module program unit.

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

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**

<table>
<thead>
<tr>
<th>OPTIONS Statements</th>
<th>USE Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statements</td>
<td></td>
</tr>
<tr>
<td>IMPLICIT NONE Statements</td>
<td></td>
</tr>
<tr>
<td>PARAMETER Statements</td>
<td>IMPLICIT Statements</td>
</tr>
<tr>
<td>NAMELIST, FORMAT, and ENTRY Statements</td>
<td></td>
</tr>
<tr>
<td>PARAMETER and DATA Statements</td>
<td>Derived-Type Definitions, Interface Blocks, Type Declaration, Statement Function, and Specification Statements</td>
</tr>
<tr>
<td>DATA Statements</td>
<td>Executable Statements</td>
</tr>
<tr>
<td>CONTAINS Statement</td>
<td></td>
</tr>
<tr>
<td>Internal Subprograms or Module Subprograms</td>
<td></td>
</tr>
<tr>
<td>END Statement</td>
<td></td>
</tr>
</tbody>
</table>

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

### Statements Restricted in Scoping Units

<table>
<thead>
<tr>
<th>Scoping Unit</th>
<th>Restricted Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main program</td>
<td>ENTRY and RETURN statements</td>
</tr>
<tr>
<td>Module ¹</td>
<td>ENTRY, FORMAT, OPTIONAL, and INTENT statements, statement functions, and executable statements</td>
</tr>
<tr>
<td>Block data program unit</td>
<td>CONTAINS, ENTRY, and FORMAT statements, interface blocks, statement functions, and executable statements</td>
</tr>
<tr>
<td>Internal subprogram</td>
<td>CONTAINS and ENTRY statements</td>
</tr>
<tr>
<td>Interface body</td>
<td>CONTAINS, DATA, ENTRY, SAVE, and FORMAT statements, statement functions, and executable statements</td>
</tr>
</tbody>
</table>

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

In Fortran 95/90, a name can contain up to 31 characters. Compaq 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 Compaq. On Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux 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

Compaq Fortran supports the following characters:

- The Fortran 95/90 character set which consists of the following:
  - All uppercase and lowercase letters (A through Z and a through z)
  - The numerals 0 through 9
  - The underscore ( _ )
  - The following special characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Character</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank or &lt;Tab&gt;</td>
<td>Blank (space) or tab</td>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>=</td>
<td>Equal sign</td>
<td>!</td>
<td>Exclamation point</td>
</tr>
<tr>
<td>+</td>
<td>Plus sign</td>
<td>&quot;</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>-</td>
<td>Minus sign</td>
<td>%</td>
<td>Percent sign</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
<td>&amp;</td>
<td>Ampersand</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
<td>;</td>
<td>Semicolon</td>
</tr>
</tbody>
</table>
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 95/90 character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records.

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

For more 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. Compaq 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**
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.

<table>
<thead>
<tr>
<th>Source Item</th>
<th>Indicator ¹</th>
<th>Source Form</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td>!</td>
<td>All forms</td>
<td>Anywhere in source code</td>
</tr>
<tr>
<td>Comment line</td>
<td>!</td>
<td>Free</td>
<td>At the beginning of the source line</td>
</tr>
<tr>
<td></td>
<td>!, C, or *</td>
<td>Fixed</td>
<td>In column 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tab</td>
<td>In column 1</td>
</tr>
<tr>
<td>Continuation line ²</td>
<td>&amp;</td>
<td>Free</td>
<td>At the end of the source line</td>
</tr>
<tr>
<td></td>
<td>Any character except zero or blank</td>
<td>Fixed</td>
<td>In column 6</td>
</tr>
<tr>
<td></td>
<td>Any digit except zero</td>
<td>Tab</td>
<td>After the first tab</td>
</tr>
<tr>
<td>Statement separator</td>
<td>;</td>
<td>All forms</td>
<td>Between statements on the same line</td>
</tr>
<tr>
<td>Statement label</td>
<td>1 to 5 decimal digits</td>
<td>Free</td>
<td>Before a statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed</td>
<td>In columns 1 through 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tab</td>
<td>Before the first tab</td>
</tr>
<tr>
<td>A debugging statement</td>
<td>D</td>
<td>Fixed</td>
<td>In column 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tab</td>
<td>In column 1</td>
</tr>
</tbody>
</table>

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.
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:

```plaintext
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

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Required Blanks</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK DATA</td>
<td>CASE DEFAULT</td>
</tr>
<tr>
<td>DO</td>
<td>DO WHILE</td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>IMPLICIT type-specifier</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>IMPLICIT NONE</td>
</tr>
<tr>
<td>END BLOCK DATA</td>
<td>INTERFACE ASSIGNMENT</td>
</tr>
<tr>
<td>END DO</td>
<td>INTERFACE OPERATOR</td>
</tr>
<tr>
<td>END FILE</td>
<td>MODULE PROCEDURE</td>
</tr>
<tr>
<td>END FORALL</td>
<td>RECURSIVE FUNCTION</td>
</tr>
<tr>
<td>END FUNCTION</td>
<td>RECURSIVE SUBROUTINE</td>
</tr>
<tr>
<td>END IF</td>
<td>RECURSIVE type-specifier FUNCTION</td>
</tr>
<tr>
<td>END INTERFACE</td>
<td>type-specifier FUNCTION</td>
</tr>
<tr>
<td>END MODULE</td>
<td>type-specifier RECURSIVE FUNCTION</td>
</tr>
<tr>
<td>END PROGRAM</td>
<td></td>
</tr>
<tr>
<td>END SELECT</td>
<td></td>
</tr>
<tr>
<td>END SUBROUTINE</td>
<td></td>
</tr>
<tr>
<td>END TYPE</td>
<td></td>
</tr>
<tr>
<td>END WHERE</td>
<td></td>
</tr>
<tr>
<td>GO TO</td>
<td></td>
</tr>
<tr>
<td>IN OUT</td>
<td></td>
</tr>
<tr>
<td>SELECT CASE</td>
<td></td>
</tr>
</tbody>
</table>

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
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, Compaq Fortran allows up to 511 continuation lines.

The following shows a continued statement:

```
TCOSH(Y) = EXP(Y) + &
          EXP(-Y) ! The initial statement line
          ! 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, Compaq 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 95/90 permits up to 19 continuation lines in a fixed-form program, Compaq 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:

```fortran
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:

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

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

---

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

For 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement label</td>
<td>1 through 5</td>
</tr>
<tr>
<td>Continuation indicator</td>
<td>6</td>
</tr>
<tr>
<td>Statement</td>
<td>7 through 72 (or 132 with the <code>/extend_source</code> compiler option)</td>
</tr>
<tr>
<td>Sequence number</td>
<td>73 through 80</td>
</tr>
</tbody>
</table>

---

**Format using TAB Character**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>F</td>
<td>I</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td></td>
<td>V</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>I = J + 5*K +</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>I = M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>IVAL = I+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Character-per-Column Format**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>F</td>
<td>I</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td></td>
<td>V</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>I</td>
<td>=</td>
<td>J</td>
<td>+</td>
<td>5 *</td>
<td>K</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>I</td>
<td>=</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>V</td>
<td>A</td>
<td>L</td>
<td>=</td>
<td>I</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
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 Compaq Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

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

For 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 Source Forms**

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

<table>
<thead>
<tr>
<th>Blanks</th>
<th>Treat as significant (see Free Source Form).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement labels</td>
<td>Place in column positions 1 through 5 (or before the first tab character).</td>
</tr>
<tr>
<td>Statements</td>
<td>Start in column position 7 (or after the first tab character).</td>
</tr>
<tr>
<td>Comment indicator</td>
<td>Use only !. Place anywhere except in column position 6 (or immediately after the first tab character).</td>
</tr>
</tbody>
</table>
The following example is valid for all source forms:

Column:

12345678...

Cont. 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).

! 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:

- Intrinsic data types and constants
- Derived data types
- Binary, octal, hexadecimal, and Hollerith constants
Variables, including arrays

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

Compaq 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, Tru64 UNIX, and Linux 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
- **COMPLEX([KIND=]16) or COMPLEX*32**
  This kind is only available on OpenVMS, Tru64 UNIX, and Linux systems.

- **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:

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

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

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

**For More Information:**
o See the KIND intrinsic function.

o On declaration statements for intrinsic data types, see Declaration Statements for Noncharacter Types and Declaration Statements for Character Types.

o On operations for intrinsic data types, see Expressions.

o 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 \textit{base} is omitted but \# is specified, the integer is interpreted in base 16. If both \textit{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.

\section*{Examples}

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

\begin{verbatim}
Valid
0
-127
+32123
47_2

Invalid  Explanation
9999999999999999999 Number too large.
3.14   Decimal point not allowed; this is a valid \textbf{REAL} constant.
32,767 Comma not allowed.
33_3   3 is not a valid kind type for integers.
\end{verbatim}

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

\begin{verbatim}
I = 2#1111001111001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
\end{verbatim}

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:

\begin{verbatim}
\begin{tabular}{l|l|l}
\hline
\textbf{Fortran Assignment} & \textbf{Integer Value in Data} & \textbf{Hexadecimal Value in Data} \\
\hline
LOGICAL(1)X  &  &  \\
INTEGER(1)X  &  &  \\
X = -128     & -128 & Z'80' \\
X = 127      & 127 & Z'7F' \\
X = 255      & -1    & Z'FF' \\
\hline
\end{tabular}
\end{verbatim}
Real Data Types

Real data types can be specified as follows:

```
REAL
REAL([KIND=]n)
REAL*\text{n}
DOUBLE PRECISION
```

\(n\)
Is kind 4, 8, or 16. Kind 16 is only available on OpenVMS, Tru64 UNIX, and Linux 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 \text{KIND} returns the kind type parameter. The intrinsic inquiry function \text{RANGE} returns the decimal exponent range, and the intrinsic function \text{PRECISION} returns the decimal precision. You can use the intrinsic function \text{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, REAL(8) or DOUBLE PRECISION Constants, and REAL(16) Constants (VMS, U*X).

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:

```fortran
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 0.0001234567, all of the nonzero digits, and none of the zeros, are significant. (See the following sections for the number of significant digits each kind type parameter typically has).

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

A real constant with no exponent part and no kind type parameter is (by default) a single-precision (REAL(4)) constant. You can change the default
behavior by specifying the compiler option `/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, Tru64 UNIX, and Linux 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, IEEE® S_floating format is used. On OpenVMS systems, either Compaq 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, IEEE T_floating format is used. On OpenVMS systems, either Compaq 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
-.25D0\_2
+2.7182812846182
123456789D45
123456789.D400
123456789.D-400

Explanation
2 is not a valid kind for reals.
No D exponent designator is present; this is a valid single-
precision constant.
Too large for D_floating format; valid for G_floating and
T_floating format.
Too large for any double-precision format.
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**

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 your user manual.

## Complex Data Types

Complex data types can be specified as follows:

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

- `n` is kind 4, 8, or 16. Kind 16 is only available on OpenVMS, Tru64 UNIX, and Linux systems.

- `s` is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

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

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

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

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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems: IEEE S_floating format
- On OpenVMS systems: Compaq 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**

(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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems: IEEE T_floating format
- On OpenVMS systems: Compaq 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**

- Second constant missing.
- Hollerith constants not allowed.
- 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.

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

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

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

The rules for REAL(16) constants apply to REAL(16) constants used in COMPLEX constants. (See General Rules for Complex Constants and REAL(16) Constants for the rules on forming REAL(16) constants.)

The REAL(16) constants in a COMPLEX constant have one of the following formats:

- On Tru64 UNIX and Linux systems: IEEE X_floating format
- On OpenVMS systems: Compaq VAX X_floating or IEEE X_floating format (depending on the compiler option specified)

Examples

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

Valid
(1.7039,-1.7039E2)
(547.3E0_16,-1.44)
(+12739D3,0.Q0)

Invalid
(1.23Q0,)
Second constant missing.
(1D1,2H12)
Hollerith constants not allowed.
(1.7039E0,-1.7039D0) Neither constant is REAL(16); this is a valid double-precision constant.

For More Information:

- See General Rules for Complex Constants.
- On the format and range of COMPLEX(16) data, see your programmer's guide.
- On compiler options affecting REAL 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:

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

An attribute-oriented example is:

```fortran
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:

```fortran
CHARACTER
CHARACTER([KIND=n])
```
\textbf{CHARACTER ([LEN=]len)}
\textbf{CHARACTER ([LEN=]len [, [KIND=}n\textbf{])}}
\textbf{CHARACTER (KIND=}n [, \textbf{LEN=}len\textbf{])}
\textbf{CHARACTER*}len[,]

\begin{itemize}
\item \textit{n}\\
Is kind 1.
\item \textit{len}\\
Is a string length (not a kind). For more information, see Declaration Statements for Character Types.
\end{itemize}

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.

\section*{Character Constants}

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

\begin{itemize}
\item \textbf{[k_]}\textbf{[ch...]} \textbf{[C]}\\
\item \textbf{[k_]"[ch...]"} \textbf{[C]}
\end{itemize}

\begin{itemize}
\item \textit{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.
\item \textit{ch}\\
Is an ASCII character.
\item \textit{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.
\end{itemize}

\section*{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? \\n'TODAY''S DATE IS: ' \\
"The average is: " \\
''

Invalid

'HEADINGS \\
'Map Number:'

Explanation

No trailing apostrophe.
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>
<thead>
<tr>
<th>C-Style Escape Sequences</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a or \A</td>
<td>A bell</td>
</tr>
<tr>
<td>\b or \B</td>
<td>A backspace</td>
</tr>
<tr>
<td>\f or \F</td>
<td>A formfeed</td>
</tr>
<tr>
<td>\n or \N</td>
<td>A new line</td>
</tr>
<tr>
<td>\r or \R</td>
<td>A carriage return</td>
</tr>
<tr>
<td>\t or \T</td>
<td>A horizontal tab</td>
</tr>
<tr>
<td>\v or \V</td>
<td>A vertical tab</td>
</tr>
<tr>
<td>\xhh or \Xhh</td>
<td>A hexadecimal bit pattern</td>
</tr>
<tr>
<td>\ooo</td>
<td>An octal bit pattern</td>
</tr>
<tr>
<td>\0</td>
<td>A null character</td>
</tr>
<tr>
<td>\</td>
<td>A backslash</td>
</tr>
</tbody>
</table>

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 \xhh 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 '\010C' and '\x08C' both represent a backspace character followed by a null character.

The C string '\\abcdC' 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:

\[
\begin{align*}
    \nu &([e1]:[e2]) \\
    \oa &([s [, s ] ... ) ([e1]:[e2]) \\
    \nu &
\end{align*}
\]

\(\nu\)
- 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 \( \text{len} \). For example, \( \text{NAMES}(1,3)(:7) \) specifies the substring starting with the first character position and ending with the seventh character position of the character array element \( \text{NAMES}(1,3) \).

**Examples**

Consider the following example:

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

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

Consider the following example:

```plaintext
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:

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

Consider the following example:

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

\( \text{BRANCH\_CHAP} \) 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:

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

Consider the following:

```fortran
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:

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

```fortran
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:

\[
\text{parent} [\%\text{component} [(s\text{-list})]] \ldots \%\text{component} [(s\text{-list})]
\]

\text{parent}

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

\text{component}

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

\text{s\text{-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 \text{s\text{-list}} must equal the rank of the immediately preceding parent or component.

\section*{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.

\section*{Examples}

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

\begin{verbatim}
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
\end{verbatim}
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:

```plaintext
TYPE EMPLOYEE
  INTEGER ID
```
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 95/90 allows unsigned binary, octal, and hexadecimal constants to be used in DATA statements; the constant must correspond to an integer scalar variable.

In Compaq 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 256 binary digits in a binary constant. Leading zeros are ignored.

Examples

The following examples demonstrate valid and invalid binary constants:

Valid
B'0101110'
B"1"

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

Octal Constants

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

O'd[d...]'
O"d[d...]"

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

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

Examples

The following examples demonstrate valid and invalid octal constants:

Valid
O'07737'
O"1"

Invalid
Explanation
O'7782'  The character 8 is invalid.
Hexadecimal Constants

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

\[ Z'd[d...]' \]
\[ Z"d[d...]" \]

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

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

Examples

The following examples demonstrate valid and invalid hexadecimal constants:

Valid

\[ Z'AF9730' \]
\[ Z"FFABC" \]
\[ Z'84' \]

Invalid

\[ Z'999.' \] 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**

3HABCD

0H

**Explanation**

Wrong number of characters.

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:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Data Type of Constant</th>
<th>Length of Constant (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(2) ICOUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER(4) JCOUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER(4) N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(8) DOUBLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(4) RAFFIA, RALPHA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAFFIA = B'1001100111111010011'</td>
<td>REAL(4)</td>
<td>4</td>
</tr>
<tr>
<td>RAFFIA = Z'99AF2'</td>
<td>REAL(4)</td>
<td>4</td>
</tr>
<tr>
<td>RALPHA = 4HABCD</td>
<td>REAL(4)</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE = B'111111111100110011010'</td>
<td>REAL(8)</td>
<td>8</td>
</tr>
<tr>
<td>DOUBLE = Z'FFF99A'</td>
<td>REAL(8)</td>
<td>8</td>
</tr>
</tbody>
</table>
When a specific data type (generally integer) is required, that type is assumed for the constant. For example:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Data Type of Constant</th>
<th>Length of Constant (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y(IX) = Y(O'15') + 3.</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>Y(IX) = Y(1HA) + 3.</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
</tbody>
</table>

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 x86 processors, a length of four bytes is used.
- For Hollerith constants, no data type is assumed.

For example:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Data Type of Constant</th>
<th>Length of Constant (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL APAC(Z'34BC2')</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>CALL APAC(9HABCDEFGHI)</td>
<td>None</td>
<td>9</td>
</tr>
</tbody>
</table>

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):
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:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Data Type of Constant</th>
<th>Length of Constant (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (Z'AF77') 1,2,3</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>IF (2HAB) 1,2,3</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>I = O'7777' - Z'A39' 1</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>I = 1HC - 1HA</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>J = .NOT. O'73777'</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>J = .NOT. 1HB</td>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
</tbody>
</table>

1. When two typeless constants are used in an operation, they both take default integer type.
- 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 is 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:
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:

<table>
<thead>
<tr>
<th>Real Variables</th>
<th>Integer Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>JCOUNT</td>
</tr>
<tr>
<td>BETA</td>
<td>ITEM_1</td>
</tr>
<tr>
<td>TOTAL_NUM</td>
<td>NTOTAL</td>
</tr>
</tbody>
</table>

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

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

**For More Information:**

- 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 a type declaration statement, or by using a DIMENSION, COMMON, ALLOCATABLE, POINTER, or TARGET statement.

An array can be referenced by element (using subscripts), by section (using a section subscript list), or as a whole. A subscript list (appended to the array name) indicates which array element or array section is being referenced.
A section subscript list consists of subscripts, subscript triplets, or vector subscripts. At least one subscript in the list must be a subscript triplet or vector subscript.

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

An array has the following properties:

- **Data type**
  
  An array can have any intrinsic or derived type. The data type of an array (like any other variable) is specified in a type declaration statement or implied by the first letter of its name. All elements of the array have the same type and kind parameters. If a value assigned to an individual array element is not the same as the type of the array, it is converted to the array's type.

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

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

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

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

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

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

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

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

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

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
...                                ! Reference to an array element
B(3) = 5.0                          ! Reference to an array section consisting of
B(2:5) = 1.0                        !   elements: B(2), B(3), B(4), B(5)
...                                ! Reference to an array element
C(4,8) = I                           ! Reference to an array section consisting of
C(1:3,3:4) = J                      !   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
```
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:

\[ \text{array(subscript-list)} \]

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:

\[ \text{ARRAY_D(1,2) (1:3)} \]

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**

**One-Dimensional Array** `BRC(6)`

```
1  BRC(1)  2  BRC(2)  3  BRC(3)  4  BRC(4)  5  BRC(5)  6  BRC(6)
```

**Two-Dimensional Array** `BAN(3,4)`

```
1      BAN(1,1)  4      BAN(1,2)  7      BAN(1,3) 10      BAN(1,4)
2      BAN(2,1)  5      BAN(2,2)  8      BAN(2,3) 11      BAN(2,4)
3      BAN(3,1)  6      BAN(3,2)  9      BAN(3,3) 12      BAN(3,4)
```

**Three-Dimensional Array** `BOS(3,3,3)`

```
19  BOS(1,1,1)  22  BOS(1,2,1)  25  BOS(1,3,1)  10  BOS(3,1,1)
20  BOS(1,1,2)  23  BOS(1,2,2)  26  BOS(1,3,2)  11  BOS(3,1,2)
13  BOS(1,1,3)  16  BOS(1,2,3)  27  BOS(1,3,3)  14  BOS(3,1,3)
```

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:

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

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

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

```plaintext
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:

  \[
  B(2,1,2) \quad B(2,2,2) \\
  B(3,1,2) \quad B(3,2,2) \\
  B(4,1,2) \quad B(4,2,2)
  \]

  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
A(2:3) = 3.0     ! The upper bound defaults to 10
A(7:9) = 3.0     ! The stride defaults to 1
A(:) = 3.0       ! Same as A = 3.0; sets all elements of A to 3.0
```

For More Information:

- See Array Sections.

Vector Subscripts

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

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

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

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

```fortran
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:

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

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

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

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

**For More Information:**

- See [Array Sections](#).
- See [Array constructors](#).

### 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:

<table>
<thead>
<tr>
<th>Form of <em>ac-value</em></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A scalar expression</td>
<td>Its value is an element of the new array.</td>
</tr>
<tr>
<td>An array expression</td>
<td>The values of the elements in the expression (in array element order) are the corresponding sequence of elements in the new array.</td>
</tr>
<tr>
<td>An implied-do loop</td>
<td>It is expanded to form a list of array elements under control of the DO variable (like a <strong>DO</strong> construct).</td>
</tr>
</tbody>
</table>

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

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

You can also mix these forms, for example:

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

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

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

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

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

This causes the following to be displayed:

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

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

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

The following are alternative forms for array constructors:

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

```fortran
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:

```fortran
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 /)
```

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
- Assignment statements

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. The default for .TRUE. is -1; .FALSE. is 0. Note that on Windows NT (including Windows 2000) and Windows 9* systems, the default can change if compiler option /fpscomp:logicals is used.

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

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

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+</td>
<td>Addition or unary plus (identity)</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or unary minus (negation)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Highest</td>
</tr>
<tr>
<td>* and /</td>
<td>.</td>
</tr>
<tr>
<td>Unary + and -</td>
<td>.</td>
</tr>
<tr>
<td>Binary + and -</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

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, Compaq 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:

\[ A^{**}B\ast C \text{ is evaluated as } (A^{**}B) \ast C \]

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

\[ A^{**} -(B\ast C) \text{ is evaluated as } A^{**}(-(B\ast C)) \]

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:

\[ X/-15.0\ast Y \text{ is evaluated as } X/-(15.0\ast Y) \]

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.

\[
\begin{align*}
4 + 3 \ast 2 - 6/2 & = 7 \\
& \quad ^\wedge \quad ^\wedge \quad ^\wedge \\
& 2 \quad 1 \quad 4 \quad 3 \\
\end{align*}
\]

\[
\begin{align*}
(4 + 3) \ast 2 - 6/2 & = 11 \\
& \quad ^\wedge \quad ^\wedge \quad ^\wedge \quad ^\wedge \\
& 1 \quad 2 \quad 4 \quad 3 \\
\end{align*}
\]

\[
\begin{align*}
(4 + 3 \ast 2 - 6)/2 & = 2 \\
\end{align*}
\]
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:

\[
4 + (3 \times 2) - (6/2)
\]

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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL(1) and BYTE</td>
<td>Lowest</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td></td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td></td>
</tr>
<tr>
<td>LOGICAL(8)(^1)</td>
<td></td>
</tr>
</tbody>
</table>
The data type of the value produced by an operation on two numeric operands of different data types is the data type of the highest-ranking operand in the operation. For example, the value resulting from an operation on an integer and a real operand is of real type. However, an operation involving a COMPLEX(4) or COMPLEX(8) data type and a DOUBLE PRECISION data type produces a COMPLEX(8) result.

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

- Integer operations: Integer operations are performed only on integer operands. (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 \( \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} \) 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 \( Y = (I /J) \times X \), 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 a COMPLEX(4) or COMPLEX(8) operand and a DOUBLE PRECISION operand are performed as COMPLEX(8) operations; the DOUBLE PRECISION operand is not rounded.

These rules also generally apply to numeric operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a DOUBLE PRECISION (REAL(8)) or REAL(16) (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:

\[
\begin{align*}
('ABC//'DE')//'F' \\
'ABC'//('DE//'F') \\
'ABC'//'DE//'F'
\end{align*}
\]

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 '//'DE//'F' has a value of 'ABC DEFIN'.

**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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT. or &lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>.LE. or &lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>.EQ. or = =</td>
<td>Equal to</td>
</tr>
<tr>
<td>.NE. or /=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>.GT. or &gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>.GE. or &gt;=</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

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:

\[
\text{APPLE+PEACH} > \text{PEAR+ORANGE}
\]

This expression states that the sum of APPLE and PEACH is greater than the
Expressions and Assignment Statements

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 (\(\text{=}\) or .EQ.) or the not equal operator (/\(\neq\) 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:
Periods cannot appear consecutively except when the second operator is .NOT. For example, the following logical expression is valid:

\[ A + B/(A-1) \text{ .AND. } .NOT. D + B/(D-1) \]

### 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>A .AND. B</td>
<td>Logical conjunction: the expression is true if both A and B are true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>A .OR. B</td>
<td>Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>A .NEQV. B</td>
<td>Logical inequivalence (exclusive OR): the expression is true if either A or B is true, but false if both are true.</td>
</tr>
<tr>
<td>.XOR.</td>
<td>A .XOR. B</td>
<td>Same as .NEQV.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>A .EQV. B</td>
<td>Logical equivalence: the expression is true if both A and B are true, or both are false.</td>
</tr>
<tr>
<td>.NOT.</td>
<td>.NOT. A</td>
<td>Logical negation: the expression is true if A is false and false if A is true.</td>
</tr>
</tbody>
</table>

\(^1\) .NOT. is a unary operator.
Expressions and Assignment Statements

\[ \text{A*B+C*ABC} = \text{X*Y+DM/ZZ .AND. .NOT. \ K*B > TT} \]

This expression is evaluated in the following sequence:

\[ (((\text{A*B})+(\text{C*ABC})) = ((\text{X*Y})+(\text{DM/ZZ}))) .\text{AND.} (\text{.NOT. } ((\text{K*B}) > TT)) \]

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 \((\text{A(I)}+1.0)\) or \(\text{B(I)}*2.0\) could be evaluated first:

\[ (\text{A(I)}+1.0) \text{.GT. } \text{B(I)}*2.0 \]

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:

\[ \text{A .AND. (F(X,Y) .GT. 2.0) .AND. B} \]

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 \text{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.
Expressions and Assignment Statements

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

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

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

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

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

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

**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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Defined Unary Operators</td>
<td>Highest</td>
</tr>
<tr>
<td>Numeric</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>* or /</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Unary + or -</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>Binary + or -</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>//</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>.EQ., .NE., .LT., .LE., .GT., .GE., =, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>.NOT.</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>.AND.</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>.OR.</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>.XOR., .EQV., .NEQV.</td>
<td></td>
</tr>
<tr>
<td>Defined Binary Operators</td>
<td>Lowest</td>
<td></td>
</tr>
</tbody>
</table>
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

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)
- A structure constructor whose components are restricted expressions
- An implied-do variable within an array constructor, where the bounds and strides of the corresponding implied-do are restricted expressions
- A reference to one of the following inquiry functions:

<table>
<thead>
<tr>
<th>BIT_SIZE</th>
<th>NWORKERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS</td>
<td>PRECISION</td>
</tr>
<tr>
<td>EPSILON</td>
<td>PROCESSORS_SHAPE</td>
</tr>
<tr>
<td>HUGE</td>
<td>RADIX</td>
</tr>
<tr>
<td>ILEN</td>
<td>RANGE</td>
</tr>
<tr>
<td>KIND</td>
<td>SHAPE</td>
</tr>
<tr>
<td>LBOUND</td>
<td>SIZE</td>
</tr>
<tr>
<td>LEN</td>
<td>SIZEOF</td>
</tr>
<tr>
<td>MAXEXponent</td>
<td>TINY</td>
</tr>
</tbody>
</table>
Each function argument must be one of the following:

- A restricted expression

- A variable whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, are not defined by an expression that is not a restricted expression, or are not definable by an `ALLOCATE` or pointer assignment statement.
  
  - A reference to any other intrinsic function where each argument is a restricted expression.
  
  - A reference to a specification function where each argument is a restricted expression

  - An array constructor where each element and the bounds and strides of each implied-do, are expressions whose primaries are restricted expressions

  - Another restricted expression enclosed in parentheses

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

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

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

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

- By a previous declaration in the same scoping unit
- By the implicit typing rules currently in effect for the scoping unit
- By host or use association
If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

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

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

Examples

The following shows valid specification expressions:

\[
\begin{align*}
\text{MAX}(I) + J & \quad \text{! I and J are scalar integer variables} \\
\text{UBOUND}(\text{ARRAY}_B, 20) & \quad \text{! ARRAY}_B \text{ is an assumed-shape dummy array}
\end{align*}
\]

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 (WHERE), and element array (FORALL).

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:

\[ \text{variable} = \text{expression} \]

**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 \(-2\times10^{9}\) to \(+2\times10^{9}\), 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) and COMPLEX(16) are only available on OpenVMS, Tru64 UNIX, and Linux systems.

<table>
<thead>
<tr>
<th>Scalar Memory Reference (V)</th>
<th>Expression (E)</th>
<th>Complex</th>
</tr>
</thead>
</table>
| Integer or Logical          | V=INT(E)       | V=INT(REAL(E))
|                             |                | Imaginary part of E is not used. |
| REAL (KIND=4)               | V=REAL(E)      | V=REAL(REAL(E))
|                             |                | Imaginary part of E is not used. |
| REAL (KIND=8)               | V=DBLE(E)      | V=DBLE(REAL(E))
|                             |                | Imaginary part of E is not used. |
| REAL (KIND=16)              | V=QEXT(E)      | V=QEXT(REAL(E))
|                             |                | Imaginary part of E is not used. |
For more information, see INT, REAL, DBLE, QEXT, CMPLX, and AIMAG.

**Examples**

The following examples demonstrate valid and invalid numeric assignment statements:

**Valid**

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

**Invalid Explanation**

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

**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'
RE Vol(1) = 'MAR'/'CIA'
LOCA(3:8) = 'PLANT5'
TEXT(I,J+1)(2:N-1) = NAME/ /X
```

**Invalid**

```

Explanation
```
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:

```fortran
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:

\[
\begin{align*}
\text{REAL } A(20) \\
... \\
A(1:20) &= A(20:1:-1)
\end{align*}
\]

**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:

```plaintext
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
Rules and Behavior

If the target is a variable, it must have the POINTER or TARGET attribute, or be a subobject whose parent object has the TARGET attribute.

If the target is an expression, the result must be a pointer.

If the target is not a pointer (it has the TARGET attribute), the pointer object is associated with the target.

If the target is a pointer (it has the POINTER attribute), its status determines the status of the pointer object, as follows:

- If the pointer is associated, the pointer object is associated with the same object as the target
- If the pointer is disassociated, the pointer object becomes disassociated
- If the pointer is undefined, the pointer object becomes undefined

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined.

When pointer assignment occurs, any previous association between the pointer object and a target is terminated.

Pointers can also be assigned for a pointer structure component by execution of a derived-type intrinsic assignment statement or a defined assignment statement.

Pointers can also become associated by using the ALLOCATE statement to allocate the pointer.

Pointers can become disassociated by deallocation, nullification of the pointer (using the DEALLOCATE or NULLIFY statements), or by reference to the NULL intrinsic function.

Examples

The following are examples of pointer assignments:

```
HOUR => MINUTES(1:60)        ! target is an array
M_YEAR => MY_CAR%YEAR        ! target is a structure component
NEW_ROW%RIGHT => CURRENT_ROW ! pointer object is a structure component
PTR => M                      ! target is a variable
POINTER_C => NULL ()          ! reference to NULL intrinsic
```
The following example shows a target as a pointer:

```
INTEGER, POINTER :: P, N
INTEGER, TARGET :: M
INTEGER S

M = 14
N => M                      ! N is associated with M
P => N                      ! P is associated with M through N
S = P + 5
```

The value assigned to S is 19 (14 + 5).

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](arrays).
- See [ALLOCATE](allocate).
- See [ASSOCIATED](associated).
- See [DEALLOCATE](deallocate).
- See [NULLIFY](nullify).
- See [NULL](null).
- See [POINTER](pointer).
- See [Defined assignments](defined).
- On derived-type intrinsic assignments, see [Intrinsic Assignments](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](where).

See also [FORALL](forall).
FORALL Statement and Construct

The **FORALL** statement and construct is a generalization of the Fortran 95/90 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 95/90, 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**
Overides 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:
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.

<table>
<thead>
<tr>
<th>Noncharacter Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE1</td>
</tr>
<tr>
<td>LOGICAL2</td>
</tr>
<tr>
<td>LOGICAL([KIND=]1) (or LOGICAL*1)</td>
</tr>
<tr>
<td>LOGICAL([KIND=]2) (or LOGICAL*2)</td>
</tr>
<tr>
<td>LOGICAL([KIND=]4) (or LOGICAL*4)</td>
</tr>
<tr>
<td>LOGICAL([KIND=]8) (or LOGICAL*8)³</td>
</tr>
<tr>
<td>INTEGER⁴</td>
</tr>
</tbody>
</table>
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:

```fortran
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=]n]])
CHARACTER [([KIND=n [, 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems; 65535 for OpenVMS systems. Negative values are treated as zero.

*n*  
Is a scalar integer initialization expression specifying a valid kind parameter. Currently the only kind available is 1.

**Rules and Behavior**

An automatic object can appear in a character declaration. The object cannot be a dummy argument, and its length must be declared with a specification expression that is not a constant expression.

The length specified for a character-valued statement function or statement function dummy argument of type character must be an integer constant expression.

When an asterisk length specification *(*) is used for a function name or dummy argument, it assumes the length of the corresponding function reference or actual argument. Similarly, when an asterisk length specification is used for a named constant, the name assumes the length of the actual constant it represents. For example, STRING assumes a 9-byte length in the following statements:

```
CHARACTER*(*) STRING
PARAMETER (STRING = 'VALUE IS: ')
```

A function name must not be declared with a * length, if the function is an internal or module function, or if it is array-valued, pointer-valued, recursive, or pure.

The form CHARACTER*(*) is an obsolescent feature in Fortran 95.

**Examples**
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 'ABCDEFGHJ'.

    CHARACTER*32 NAMES(100),SOCSEC(100)*9,NAMETY*10 /'ABCDEFGHJ'/

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:

\[(a\text{-spec})\]

\[a\text{-spec}\]

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:

```fortran
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
END SUBROUTINE SUB
```

**For More Information:**

For details on the general form and rules for type declaration statements, see Type Declarations in the *A to Z Reference*.

**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:

```fortran
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:

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

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:

```fortran
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:

```fortran
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:

```fortran
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:

```plaintext
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:

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

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:

```plaintext
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:

\[
([\text{expli-shape-spec}], [\text{expli-shape-spec}], ... [dl:] *)
\]

*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:

```fortran
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:

```fortran
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:

```fortran
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:

```fortran
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:

```fortran
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 EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following example:

```plaintext
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**
Each of the following statements also aligns the two arrays as shown in the above table:

\[
\begin{align*}
\text{EQUIVALENCE} & \left( \text{TABLE, TRIPLE}(2,2,1) \right) \\
\text{EQUIVALENCE} & \left( \text{TRIPLE}(1,1,2), \text{TABLE}(2,1) \right)
\end{align*}
\]

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:

\[
\text{EQUIVALENCE} \left( A(3,4), B(2,4) \right)
\]

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:

\[
\begin{align*}
\text{EQUIVALENCE} & \left( A, B(4,1) \right) \\
\text{EQUIVALENCE} & \left( B(3,2), A(2,2) \right)
\end{align*}
\]

**Equivalence of Arrays with Nonunity Lower Bounds**
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:

\[
\text{EQUIVALENCE (NAME(9:9), ID(1:1))}
\]

**Equivalence of Substrings**

<table>
<thead>
<tr>
<th>NAME</th>
<th>Character Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Character Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

ZK-0618-GE

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).

\[
\text{CHARACTER FIELDS(100)*4, STAR(5)*5}
\]

\[
\text{EQUIVALENCE (FIELDS(1)(2:4), STAR(2)(3:5))}
\]

**Equivalence of Character Arrays**

<table>
<thead>
<tr>
<th>STAR</th>
<th>Character Position</th>
<th>Subscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIELDS</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
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**

Valid

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

```
<table>
<thead>
<tr>
<th>A(1)</th>
<th>A(2)</th>
<th>A(3)</th>
<th>A(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3)</td>
<td>B(4)</td>
</tr>
</tbody>
</table>
```

```
Existing Common  Extended Portion
```

ZK-1944-GE

**An Invalid Extension of a Common Block**

Invalid

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

```
<table>
<thead>
<tr>
<th>A(1)</th>
<th>A(2)</th>
<th>A(3)</th>
<th>A(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3)</td>
<td>B(4)</td>
</tr>
</tbody>
</table>
```

```
Extended Portion  Existing Common  Extended Portion
```

ZK-1945-GE

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 95/90, pointers, allocatable arrays, and automatic arrays are dynamic data objects.

No storage space is created for a pointer until it is allocated with an `ALLOCATE` statement or until it is assigned to a allocated target. A pointer can be dynamically disassociated from a target by using a `NULLIFY` statement.

An `ALLOCATE` statement can also be used to create storage for an allocatable array. A `DEALLOCATE` statement is used to free the storage space reserved in a previous `ALLOCATE` statement.

Automatic arrays differ from allocatable arrays in that they are automatically allocated and deallocated whenever you enter or leave a procedure, respectively.

---

**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:

- The **ALLOCATE Statement**
- The **DEALLOCATE Statement**
- The **NULLIFY Statement**

**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 95/90 standard-conforming statements instead of calling an operating system memory allocation routine.

Allocating Virtual Memory

! Program accepts an integer and displays square root values

```fortran
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 \texttt{ASSOCIATED} can be used to determine whether a pointer is currently associated with a target. (The association status of the pointer must be \textit{defined}.) For example:

\begin{verbatim}
REAL, TARGET :: TAR(0:50)
REAL, POINTER :: PTR(:)
PTR => TAR
... IF (ASSOCIATED(PTR,TAR))... 
\end{verbatim}

For More Information:

- See \texttt{POINTER}.
- See \texttt{Pointer assignments}.
- See \texttt{ASSOCIATED}.

\textbf{DEALLOCATE Statement}

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

This section also discusses the following:

- Deallocation of Allocatable Arrays
- Deallocation of Pointer Targets

\textbf{Deallocation of Allocatable Arrays}

If the \texttt{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 \texttt{RETURN} or \texttt{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:

```fortran
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
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 95/90, control constructs (CASE, DO, and IF) can be named. The name must be a unique identifier in the scoping unit, and must appear on the initial line and terminal line of the construct. On the initial line, the name is separated from the statement keyword by a colon (:).

A block can contain any executable Fortran statement except an END statement. You can transfer control out of a block, but you cannot transfer control into another block.

DO loops cannot partially overlap blocks. The DO statement and its terminal statement must appear together in a statement block.

This chapter contains information on the following topics:

- Branch statements
- The CALL statement
- The CASE construct
- The CONTINUE statement
- The DO construct
- The END statement
- The IF construct and statement
- The PAUSE statement
- The RETURN statement
- The STOP statement

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:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DO terminal statement</strong></td>
<td>The branch must be taken from within its nonblock <strong>DO</strong> construct(^1).</td>
</tr>
<tr>
<td><strong>END DO</strong></td>
<td>The branch must be taken from within its block <strong>DO</strong> construct.</td>
</tr>
<tr>
<td><strong>END IF</strong></td>
<td>The branch should be taken from within its <strong>IF</strong> construct(^2).</td>
</tr>
<tr>
<td><strong>END SELECT</strong></td>
<td>The branch must be taken from within its <strong>CASE</strong> construct.</td>
</tr>
</tbody>
</table>

\(^1\) If the terminal statement is shared by more than one nonblock **DO** construct, the branch can only be taken from within the innermost **DO** construct.

\(^2\) You can branch to an **END IF** statement from outside the **IF** construct; this is a deleted feature in Fortran 95. Compaq Fortran fully supports features deleted in Fortran 95.

The following branch statements are described in this section:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO (the **ASSIGN** statement is also described here)
- Arithmetic IF

**For More Information:**

- 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.

The ASSIGN and assigned GO TO statements are deleted features in Fortran 95; they were obsolescent features in Fortran 90. Compaq Fortran fully supports features deleted in Fortran 95.

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

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:

- For simple **DO** constructs, there is no loop control. Statements in the **DO** range are repeated until the **DO** statement is terminated explicitly by a statement within the range.

- For iterative **DO** statements, loop control is specified as `do-var = expr1, expr2 [,expr3]`. An iteration count specifies the number of times the **DO** range is executed. (For more information, see **Iteration Loop Control**.)

- For **DO WHILE** statements, loop control is specified as a **DO** range. The **DO** range is repeated as long as a specified condition remains true. Once the condition is evaluated as false, the **DO** construct terminates. (For more information, see the **DO WHILE** statement.)

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 a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Compaq 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:

   \[
   \text{MAX} (\text{INT} \left( \frac{\text{expr2} - \text{expr1} + \text{expr3}}{\text{expr3}} \right), 0) \]

   The iteration count is zero if either of the following is true:

   \[
   \begin{align*}
   \text{expr1} > \text{expr2} & \quad \text{and} \quad \text{expr3} > 0 \\
   \text{expr1} < \text{expr2} & \quad \text{and} \quad \text{expr3} < 0
   \end{align*}
   \]

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 95 and Fortran 90, 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**

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*.

The **PAUSE** statement is a deleted feature in Fortran 95; it was an obsolescent feature in Fortran 90. Compaq Fortran fully supports features deleted in Fortran 95.

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 95/90 program consists of one or more program units. There are four types of program units:

- Main program
  
The program unit that denotes the beginning of execution. It may or may not have a `PROGRAM` statement as its first statement.

- External procedures
  
  Program units that are either user-written functions or subroutines.

- Modules
  
  Program units that contain declarations, type definitions, procedures, or interfaces that can be shared by other program units.

- Block data program units
  
  Program units that provide initial values for variables in named common blocks.

A program unit does not have to contain executable statements; for example, it can be a module containing interface blocks for subroutines.

A procedure can be invoked during program execution to perform a specific task. There are several kinds of procedures, as follows:

<table>
<thead>
<tr>
<th>Kind of Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Procedure</td>
<td>A procedure that is not part of any other program unit.</td>
</tr>
<tr>
<td>Module Procedure</td>
<td>A procedure defined within a module</td>
</tr>
<tr>
<td>Internal Procedure(^1)</td>
<td>A procedure (other than a statement function) contained within a main program, function, or subroutine</td>
</tr>
<tr>
<td>Intrinsic Procedure</td>
<td>A procedure defined by the Fortran language</td>
</tr>
</tbody>
</table>
A *function* is invoked in an expression using the name of the function or a defined operator. It returns a a single value (function result) that is used to evaluate the expression.

A *subroutine* is invoked in a **CALL** statement or by a defined assignment statement. It does not directly return a value, but values can be passed back to the calling program unit through arguments (or variables) known to the calling program.

Recursion (direct or indirect) is permitted for functions and subroutines.

A procedure interface refers to the properties of a procedure that interact with or are of concern to the calling program. A procedure interface can be explicitly defined in interface blocks. All program units, except block data program units, can contain interface blocks.

This chapter contains information on the following topics:

- **Main program**
- **Modules and module procedures**
- **Block data program units**
- **Functions, subroutines, and statement functions**
- **External procedures**
- **Internal procedures**
- **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 its name 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 program unit references a module in a **USE** statement. This module reference lets the program unit access the public definitions, specifications, and procedures in the module.

Entities in a module are public by default, unless the **USE** statement specifies otherwise or the PRIVATE attribute is specified for the module entities.

A module reference causes use association between the using program unit and the entities in the module.

**For More Information:**

- 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:

```fortran
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:

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Defining Statements</th>
<th>Control Transfer Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>FUNCTION or ENTRY</td>
<td>Function reference¹</td>
</tr>
<tr>
<td>Subroutine</td>
<td>SUBROUTINE or ENTRY</td>
<td>CALL statement²</td>
</tr>
<tr>
<td>Statement function</td>
<td>Statement function definition</td>
<td>Function reference</td>
</tr>
</tbody>
</table>

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
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.

Recursive Procedures

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:

```plaintext
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](#).
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 are user-written functions or subroutines. They are located outside of the main program and can't be part of any other program unit.

External procedures can be invoked by the main program or any procedure of an executable program.

In Fortran 95/90, external procedures can include internal 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

Internal procedures are functions or subroutines that follow a CONTAINS statement in a program unit. The program unit in which the internal procedure
appears is called its *host*.

Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram.

An internal procedure takes the following form:

```
CONTAINS
  internal-subprogram
  [internal-subprogram] ...
```

*internal-subprogram*  
Is a function or subroutine subprogram that defines the procedure. An internal subprogram must not contain any other internal subprograms.

**Rules and Behavior**

Internal procedures are the same as external procedures, except for the following:

- Only the host program unit can use an internal procedure.
- An internal procedure has access to host entities by host association; that is, names declared in the host program unit are useable within the internal procedure.
- In Fortran 95/90, the name of an internal procedure must not be passed as an argument to another procedure. However, Compaq Fortran allows an internal procedure name to be passed as an actual argument to another procedure.
- An internal procedure must not contain an **ENTRY** statement.

An internal procedure can reference itself (directly or indirectly); it can be referenced in the execution part of its host and in the execution part of any internal procedure contained in the same host (including itself).

The interface of an internal procedure is always explicit.

**Examples**

The following example shows an internal procedure:

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

```fortran
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:

```fortran
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:

```fortran
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:

```fortran
PROGRAM RESULT
TEST_RESULT = LGFUNC(A, B=D)
...
END
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:

```fortran
! 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.
Array Arguments

Arrays are sequences of elements. Each element of an actual array is associated with the element of the dummy array that has the same position in array element order.

If the dummy argument is an explicit-shape or assumed-size array, the size of the dummy argument array must not exceed the size of the actual argument array.

The type and kind parameters of an explicit-shape or assumed-size dummy argument must match the type and kind parameters of the actual argument, but their ranks need not match.

If the dummy argument is an assumed-shape array, the size of the dummy argument array is equal to the size of the actual argument array. The associated actual argument must not be an assumed-size array or a scalar (including a designator for an array element or an array element substring).

If the actual argument is an array section with a vector subscript, the associated dummy argument must not be defined.

The declaration of an array used as a dummy argument can specify the lower bound of the array.

Although most types of arrays can be used as dummy arguments, allocatable arrays cannot be dummy arguments. Allocatable arrays can be used as actual arguments.

Dummy argument arrays declared as assumed-shape, deferred-shape, or pointer arrays require an explicit interface visible to the caller.

For More Information:

- 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, 4HABC\text{D}), the corresponding dummy argument must have a numeric data type.

The following example shows character and Hollerith constants being used as actual arguments:

\[
\text{SUBROUTINE S(CHARSUB, HOLLSUB, A, B)}  \\
\quad \text{EXTERNAL CHARSUB, HOLLSUB}  \\
\quad \ldots  \\
\quad \text{CALL CHARSUB(A, 'STRING')}  \\
\quad \text{CALL HOLLSUB(B, 6HSTRING)}
\]

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. They cause execution to transfer to a labeled statement rather than to the statement immediately following the statement that called the routine. The alternate return is indicated by an asterisk (*). (An alternate return is an [obsolescent](#) feature in Fortran 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:

\[
\text{CALL MINN(X, Y, *300, *250, Z)}
\]
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:

```fortran
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 six specific intrinsic functions that calculate cosines: **COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS**. These functions return different values: REAL(4), REAL(8), REAL(16) (VMS, U*X), COMPLEX(4), COMPLEX(8), and COMPLEX(16) (VMS, U*X) 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

```fortran
PROGRAM SINES
  DOUBLE PRECISION X, PI
  PARAMETER (PI=3.141592653589793238D0)
  COMMON V(3)

  ! Define SIN as a statement function
  SIN(X) = COS(PI/2-X)
  DO X = -PI, PI, 2*PI/100
    ! Reference the statement function SIN
    WRITE (6,100) X, V, SIN(X)
  END DO
  CALL COMPUT(X)
100    FORMAT (5F10.7)
END
```

```fortran
SUBROUTINE COMPUT(Y)
  DOUBLE PRECISION Y

  ! Use intrinsic function SIN as an actual argument
  INTRINSIC SIN
  COMMON V(3)

  ! Define generic reference to double-precision sine
  V(1) = SIN(Y)

  ! Use intrinsic function SIN as an actual argument
  CALL SUB(REAL(Y),SIN)
END
```

```fortran
SUBROUTINE SUB(A,S)
  ! Declare SIN as name of a user function
  EXTERNAL SIN
```

! Declare SIN as type DOUBLE PRECISION

DOUBLE PRECISION SIN
COMMON V(3)

! Evaluate intrinsic function SIN

V(2) = S(A)

! Evaluate user-defined SIN function

V(3) = SIN(A)
END

! Define the user SIN function

DOUBLE PRECISION FUNCTION SIN(X)
INTEGER FACTOR
SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5) & - X**7/FACTOR(7)
END

INTEGER FUNCTION FACTOR(N)
FACTOR = 1
DO I=N,1,-1
   FACTOR = FACTOR * I
END DO
END

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.

The statement function SIN is called.

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

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

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

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

The type of SIN is declared double precision.

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

The user-defined SIN function is evaluated.
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:

```fortran
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, Compaq 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

Every procedure has an interface, which consists of the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units.

If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration). The following table shows which procedures have implicit or explicit interfaces:

<table>
<thead>
<tr>
<th>Kind of Procedure</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>External procedure</td>
<td>Implicit ¹</td>
</tr>
<tr>
<td>Module Procedure</td>
<td>Explicit</td>
</tr>
<tr>
<td>Internal Procedure</td>
<td>Explicit</td>
</tr>
<tr>
<td>Intrinsic Procedure</td>
<td>Explicit</td>
</tr>
<tr>
<td>Dummy Procedure</td>
<td>Implicit ¹</td>
</tr>
<tr>
<td>Statement function</td>
<td>Implicit</td>
</tr>
</tbody>
</table>

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:

```fortran
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](#).
- See [Pure procedures](#).
- See [Elemental procedures](#).
- 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:

```fortran
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:

```fortran
INTEGER V1, V2
CALL GROUP_SUBS (V1, V2)
```

Consider the following:

```fortran
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)
```

where

- *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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined Unary</td>
<td>.defined-operator. operand¹</td>
<td>.MINUS. C</td>
</tr>
<tr>
<td>Defined Binary</td>
<td>operand² .defined-operator. operand³</td>
<td>B .MINUS. C</td>
</tr>
</tbody>
</table>
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 $\geq$) defines both forms ($\geq$ 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')
```
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) and COMPLEX(16) (VMS, U*X)

Group 3  Type default character (with identical lengths)
All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names with the same data type. All associated names with different data types become undefined.

If RESULT is specified in the ENTRY statement and RECURSIVE is specified in the FUNCTION statement, the interface of the function defined by the ENTRY statement is explicit within the function subprogram.

Examples

The following example shows a function subprogram that computes the hyperbolic functions SINH, COSH, and TANH:

```fortran
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:

```fortran
PROGRAM TEST
```
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 95/90 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:
  
  ```fortran
  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.

- Inquiry functions

- Transformational functions
These functions have one or more array-valued dummy or actual arguments, an array result, or both. The intrinsic function is not applied elementally to an array-valued actual argument; instead it changes (transforms) the argument array into another array.

- Nonelemental procedures

These procedures must be called with only scalar arguments; they return scalar results. All subroutines (except MVBITS) are nonelemental.

Intrinsic procedures are invoked the same way as other procedures, and follow the same rules of argument association.

The intrinsic procedures have generic (or common) names, and many of the intrinsic functions have specific names. (Some intrinsic functions are both generic and specific.)

In general, generic functions accept arguments of more than one data type; the data type of the result is the same as that of the arguments in the function reference. For elemental functions with more than one argument, all arguments must be of the same type (except for the function MERGE).

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Some specific intrinsic functions are not allowed as actual arguments in all circumstances. 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.

**Examples:**

The syntax for the `DATE_AND_TIME` intrinsic subroutine shows four optional positional arguments: `DATE`, `TIME`, `ZONE`, and `VALUES`. The following shows some valid ways to specify these arguments:

```plaintext
! Keyword example
CALL DATE_AND_TIME (ZONE=Z)

! The following two positional examples are equivalent:
CALL DATE_AND_TIME (DATE, TIME, ZONE)
CALL DATE_AND_TIME (, , ZONE)
```

For More Information:

- On argument keywords in subroutine references, see `CALL`.
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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Computation</td>
<td>Elemental functions that perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DPROD, DREAL, FLOOR, IFIX, IMAG, INT, MAX, MAX1, MIN, MIN1, MOD, MODULO, NINT, QCMPLEX, QEXT, QFLOAT, QREAL, RAN, REAL, SIGN, SNGL, ZEXT</td>
</tr>
<tr>
<td></td>
<td>Manipulation</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>Inquiry</td>
<td>Functions that return scalar values from the models associated with the type and kind parameters of their arguments: DIGITS, EPSILON, HUGE, ILEN, MAXEXPONENT, MINEXPONENT, PRECISION, RADIUS, RANGE, SIZEOF, TINY</td>
</tr>
</tbody>
</table>

- On argument keywords in function references, see Function references.
- See Argument Association.
<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformational</td>
<td>Functions that perform vector and matrix multiplication: DOT_PRODUCT, MATMUL</td>
</tr>
<tr>
<td>System</td>
<td>Functions that return information about a process or processor: PROCESSORS_SHAPE, NWORKERS, NUMBER_OF_PROCESSORS, SECNDS</td>
</tr>
<tr>
<td>Kind type</td>
<td>Functions that return kind type parameters: KIND, SELECTED_INT_KIND, SELECTED_REAL_KIND</td>
</tr>
<tr>
<td>Mathematical</td>
<td>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</td>
</tr>
<tr>
<td>Bit Manipulation</td>
<td>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</td>
</tr>
<tr>
<td>Inquiry</td>
<td>Function that lets you determine parameter s (the bit size) in the bit model: BIT_SIZE</td>
</tr>
<tr>
<td>Representation</td>
<td>Functions that return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ</td>
</tr>
<tr>
<td>Character Compare</td>
<td>Elemental functions that make a lexical comparison of the character-string arguments and return a default logical result: LGE, LGT, LLE, LLT</td>
</tr>
<tr>
<td>Conversion</td>
<td>Elemental functions that take character arguments and return integer, ASCII, or character values: ACHAR, CHAR, IACHAR, ICHAR</td>
</tr>
<tr>
<td>String handling</td>
<td>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</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Inquiry</td>
<td>Function that returns length of argument: LEN</td>
</tr>
<tr>
<td>Array</td>
<td>Functions that construct new arrays from the elements of existing array: MERGE, PACK, SPREAD, UNPACK</td>
</tr>
<tr>
<td>Construction</td>
<td>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</td>
</tr>
<tr>
<td>Inquiry</td>
<td>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</td>
</tr>
<tr>
<td>Location</td>
<td>Transformational functions that find the geometric locations of the maximum and minimum values of an array: MAXLOC, MINLOC</td>
</tr>
<tr>
<td>Manipulation</td>
<td>Transformational functions that shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE</td>
</tr>
<tr>
<td>Reduction</td>
<td>Transformational functions that perform operations on arrays. The functions &quot;reduce&quot; 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</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Functions that do the following:</td>
</tr>
<tr>
<td></td>
<td>• Let you use assembler instructions in an executable program (ASM)</td>
</tr>
<tr>
<td></td>
<td>• Check for pointer association (ASSOCIATED)</td>
</tr>
<tr>
<td></td>
<td>• Check for end-of-file (EOF)</td>
</tr>
<tr>
<td></td>
<td>• Return the class of a floating-point</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Generic Function</th>
<th>Class</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (A)</td>
<td>E</td>
<td>The absolute value of an argument</td>
</tr>
<tr>
<td>ACHAR (I)</td>
<td>E</td>
<td>The character in the specified position of the ASCII character set</td>
</tr>
<tr>
<td>ACOS (X)</td>
<td>E</td>
<td>The arc cosine (in radians) of the argument</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>ACOSD (X)</td>
<td>E</td>
<td>The arc cosine (in degrees) of the argument</td>
</tr>
<tr>
<td>ADJUSTL (STRING)</td>
<td>E</td>
<td>The specified string with leading blanks removed and placed at the end of the string</td>
</tr>
<tr>
<td>ADJUSTR (STRING)</td>
<td>E</td>
<td>The specified string with trailing blanks removed and placed at the beginning of the string</td>
</tr>
<tr>
<td>AIMAG (Z)</td>
<td>E</td>
<td>The imaginary part of a complex argument</td>
</tr>
<tr>
<td>AINT (A [,KIND])</td>
<td>E</td>
<td>A real value truncated to a whole number</td>
</tr>
<tr>
<td>ALL (MASK [,DIM])</td>
<td>T</td>
<td>.TRUE. if all elements of the masked array are true</td>
</tr>
<tr>
<td>ALLOCATED (ARRAY)</td>
<td>I</td>
<td>The allocation status of the argument array</td>
</tr>
<tr>
<td>AMAX0 (A1, A2 [, A3, ...])</td>
<td>E</td>
<td>The maximum value in a list of integers (returned as a real value)</td>
</tr>
<tr>
<td>AMIN0 (A1, A2 [, A3, ...])</td>
<td>E</td>
<td>The minimum value in a list of integers (returned as a real value)</td>
</tr>
<tr>
<td>AND (I, J)</td>
<td>E</td>
<td>See IAND</td>
</tr>
<tr>
<td>ANINT (A [, KIND])</td>
<td>E</td>
<td>A real value rounded to a whole number</td>
</tr>
<tr>
<td>ANY (MASK [,DIM])</td>
<td>T</td>
<td>.TRUE. if any elements of the masked array are true</td>
</tr>
<tr>
<td>ASIN (X)</td>
<td>E</td>
<td>The arc sine (in radians) of the argument</td>
</tr>
<tr>
<td>ASIND (X)</td>
<td>E</td>
<td>The arc sine (in degrees) of the argument</td>
</tr>
<tr>
<td>ASM (STRING [,A,...])</td>
<td>N</td>
<td>A value stored in the appropriate register by the user</td>
</tr>
<tr>
<td>Intrinsic Procedure</td>
<td>Argument Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ASSOCIATED (POINTER [,TARGET])</td>
<td>I</td>
<td>.TRUE. if the pointer argument is associated or the pointer is associated with the specified target</td>
</tr>
<tr>
<td>ATAN (X)</td>
<td>E</td>
<td>The arc tangent (in radians) of the argument</td>
</tr>
<tr>
<td>ATAND (X)</td>
<td>E</td>
<td>The arc tangent (in degrees) of the argument</td>
</tr>
<tr>
<td>ATAN2 (Y, X)</td>
<td>E</td>
<td>The inverse arc tangent (in radians) of the arguments</td>
</tr>
<tr>
<td>ATAN2D (Y, X)</td>
<td>E</td>
<td>The inverse arc tangent (in degrees) of the arguments</td>
</tr>
<tr>
<td>BIT_SIZE (I)</td>
<td>I</td>
<td>Returns the number of bits (s) in the bit model</td>
</tr>
<tr>
<td>BTEST (I, POS)</td>
<td>E</td>
<td>.TRUE. if the specified position of argument I is one</td>
</tr>
<tr>
<td>CEILING (A [,KIND])</td>
<td>E</td>
<td>The smallest integer greater than or equal to the argument value</td>
</tr>
<tr>
<td>CHAR (I [,KIND])</td>
<td>E</td>
<td>The character in the specified position of the processor character set</td>
</tr>
<tr>
<td>CMPLX (X [,Y] [,KIND])</td>
<td>E</td>
<td>The corresponding complex value of the argument</td>
</tr>
<tr>
<td>CONJG (Z)</td>
<td>E</td>
<td>The conjugate of a complex number</td>
</tr>
<tr>
<td>COS (X)</td>
<td>E</td>
<td>The cosine (in radians) of the argument</td>
</tr>
<tr>
<td>COSD (X)</td>
<td>E</td>
<td>The cosine (in degrees) of the argument</td>
</tr>
<tr>
<td>COSH (X)</td>
<td>E</td>
<td>The hyperbolic cosine of the argument</td>
</tr>
<tr>
<td>COTAN (X)</td>
<td>E</td>
<td>The cotangent (in radians) of the argument</td>
</tr>
<tr>
<td>COTAND (X)</td>
<td>E</td>
<td>The cotangent (in degrees) of the argument</td>
</tr>
<tr>
<td>COUNT (MASK [,DIM])</td>
<td>T</td>
<td>The number of .TRUE. elements in the argument array</td>
</tr>
<tr>
<td>Function</td>
<td>Result Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CSHIFT (ARRAY, SHIFT [,DIM])</td>
<td>T</td>
<td>An array that has the elements of the argument array circularly shifted</td>
</tr>
<tr>
<td>DBLE (A)</td>
<td>E</td>
<td>The corresponding double precision value of the argument</td>
</tr>
<tr>
<td>DCMPLX (X, Y)</td>
<td>E</td>
<td>The corresponding double complex value of the argument</td>
</tr>
<tr>
<td>DFLOAT (A)</td>
<td>E</td>
<td>The corresponding double precision value of the integer argument</td>
</tr>
<tr>
<td>DIGITS (X)</td>
<td>I</td>
<td>The number of significant digits in the model for the argument</td>
</tr>
<tr>
<td>DIM (X, Y)</td>
<td>E</td>
<td>The positive difference between the two arguments</td>
</tr>
<tr>
<td>DOT_PRODUCT (VECTOR_A, VECTOR_B)</td>
<td>T</td>
<td>The dot product of two rank-one arrays (also called a vector multiply function)</td>
</tr>
<tr>
<td>DREAL (A)</td>
<td>E</td>
<td>The corresponding double-precision value of the double complex argument</td>
</tr>
<tr>
<td>EOF (A)</td>
<td>I</td>
<td>.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record</td>
</tr>
<tr>
<td>EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])</td>
<td>T</td>
<td>An array that has the elements of the argument array end-off shifted</td>
</tr>
<tr>
<td>EPSILON (X)</td>
<td>I</td>
<td>The number that is almost negligible when compared to one</td>
</tr>
<tr>
<td>EXP (X)</td>
<td>E</td>
<td>The exponential value for the argument</td>
</tr>
<tr>
<td>EXPONENT (X)</td>
<td>E</td>
<td>The value of the exponent part of a real argument</td>
</tr>
<tr>
<td>FLOAT (X)</td>
<td>E</td>
<td>The corresponding real value of the integer argument</td>
</tr>
<tr>
<td>FLOOR (A [,KIND])</td>
<td>E</td>
<td>The largest integer less than or equal to the argument value</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FP_CLASS (X)</td>
<td>E</td>
<td>The class of the IEEE floating-point argument</td>
</tr>
<tr>
<td>FRACTION (X)</td>
<td>E</td>
<td>The fractional part of a real argument</td>
</tr>
<tr>
<td>HUGE (X)</td>
<td>I</td>
<td>The largest number in the model for the argument</td>
</tr>
<tr>
<td>IACHAR (C)</td>
<td>E</td>
<td>The position of the specified character in the ASCII character set</td>
</tr>
<tr>
<td>IAND (I, J)</td>
<td>E</td>
<td>The logical AND of the two arguments</td>
</tr>
<tr>
<td>IBCLR (I, POS)</td>
<td>E</td>
<td>The specified position of argument I cleared (set to zero)</td>
</tr>
<tr>
<td>IBCHNG (I, POS)</td>
<td>E</td>
<td>The reversed value of a specified bit</td>
</tr>
<tr>
<td>IBITS (I, POS, LEN)</td>
<td>E</td>
<td>The specified substring of bits of argument I</td>
</tr>
<tr>
<td>IBSET (I, POS)</td>
<td>E</td>
<td>The specified bit in argument I set to one</td>
</tr>
<tr>
<td>ICHAR (C)</td>
<td>E</td>
<td>The position of the specified character in the processor character set</td>
</tr>
<tr>
<td>IEOR (I, J)</td>
<td>E</td>
<td>The logical exclusive OR of the corresponding bit arguments</td>
</tr>
<tr>
<td>IFIX (X)</td>
<td>E</td>
<td>The corresponding integer value of the real argument rounded as if it were an implied conversion in an assignment</td>
</tr>
<tr>
<td>ILEN (I)</td>
<td>I</td>
<td>The length (in bits) in the two's complement representation of an integer</td>
</tr>
<tr>
<td>IMAG (Z)</td>
<td>E</td>
<td>See AIMAG</td>
</tr>
<tr>
<td>INDEX (STRING, SUBSTRING [,BACK])</td>
<td>E</td>
<td>The position of the specified substring in a character expression</td>
</tr>
<tr>
<td>INT (A [,KIND])</td>
<td>E</td>
<td>The corresponding integer value (truncated) of the argument</td>
</tr>
<tr>
<td>IOR (I, J)</td>
<td>E</td>
<td>The logical inclusive OR of the corresponding bit arguments</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>ISHA (I, SHIFT)</strong></td>
<td>E</td>
<td>Argument I shifted left or right by a specified number of bits</td>
</tr>
<tr>
<td><strong>ISHC (I, SHIFT)</strong></td>
<td>E</td>
<td>Argument I rotated left or right by a specified number of bits</td>
</tr>
<tr>
<td><strong>ISHFT (I, SHIFT)</strong></td>
<td>E</td>
<td>The logical end-off shift of the bits in argument I</td>
</tr>
<tr>
<td><strong>ISHFTC (I, SHIFT [,SIZE])</strong></td>
<td>E</td>
<td>The logical circular shift of the bits in argument I</td>
</tr>
<tr>
<td><strong>ISHL (I, SHIFT)</strong></td>
<td>E</td>
<td>Argument I logically shifted left or right by a specified number of bits</td>
</tr>
<tr>
<td><strong>ISNAN (X)</strong></td>
<td>E</td>
<td>Tests for Not-a-Number (NaN) values</td>
</tr>
<tr>
<td><strong>KIND (X)</strong></td>
<td>I</td>
<td>The kind type parameter of the argument</td>
</tr>
<tr>
<td><strong>LBOUND (ARRAY [,DIM [,KIND]])</strong></td>
<td>I</td>
<td>The lower bounds of an array (or one of its dimensions)</td>
</tr>
<tr>
<td><strong>LEADZ (I)</strong></td>
<td>E</td>
<td>The number of leading zero bits in an integer</td>
</tr>
<tr>
<td><strong>LEN (STRING [,KIND])</strong></td>
<td>I</td>
<td>The length (number of characters) of the argument character string</td>
</tr>
<tr>
<td><strong>LEN_TRIM (STRING)</strong></td>
<td>E</td>
<td>The length of the specified string without trailing blanks</td>
</tr>
<tr>
<td><strong>LGE (STRING_A, STRING_B)</strong></td>
<td>E</td>
<td>A logical value determined by a &gt; or = comparison of the arguments</td>
</tr>
<tr>
<td><strong>LGT (STRING_A, STRING_B)</strong></td>
<td>E</td>
<td>A logical value determined by a &gt; comparison of the arguments</td>
</tr>
<tr>
<td><strong>LLE (STRING_A, STRING_B)</strong></td>
<td>E</td>
<td>A logical value determined by a &lt; or = comparison of the arguments</td>
</tr>
<tr>
<td><strong>LLT (STRING_A, STRING_B)</strong></td>
<td>E</td>
<td>A logical value determined by a &lt; comparison of the arguments</td>
</tr>
<tr>
<td><strong>LOC (A)</strong></td>
<td>I</td>
<td>The internal address of the argument.</td>
</tr>
<tr>
<td><strong>LOG (X)</strong></td>
<td>E</td>
<td>The natural logarithm of the argument</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LOG10 (X)</td>
<td>E</td>
<td>The common logarithm (base 10) of the argument</td>
</tr>
<tr>
<td>LOGICAL (L [,KIND])</td>
<td>E</td>
<td>The logical value of the argument converted to a logical of type KIND</td>
</tr>
<tr>
<td>LSHIFT (I, POSITIVE_SHIFT)</td>
<td>E</td>
<td>See ISHFT</td>
</tr>
<tr>
<td>MATMUL (MATRIX_A, MATRIX_B)</td>
<td>T</td>
<td>The result of matrix multiplication (also called a matrix multiply function)</td>
</tr>
<tr>
<td>MAX (A1, A2 [, A3,...])</td>
<td>E</td>
<td>The maximum value in the set of arguments</td>
</tr>
<tr>
<td>MAX1 (A1, A2 [, A3,...])</td>
<td>E</td>
<td>The maximum value in the set of real arguments (returned as an integer)</td>
</tr>
<tr>
<td>MAXEXponent (X)</td>
<td>I</td>
<td>The maximum exponent in the model for the argument</td>
</tr>
<tr>
<td>MAXLOC (ARRAY [,DIM] [,MASK])</td>
<td>T</td>
<td>The rank-one array that has the location of the maximum element in the argument array</td>
</tr>
<tr>
<td>MAXVAL (ARRAY [,DIM] [,MASK])</td>
<td>T</td>
<td>The maximum value of the elements in the argument array</td>
</tr>
<tr>
<td>MERGE (TSOURCE, FSOURCE, MASK)</td>
<td>E</td>
<td>An array that is the combination of two conformable arrays (under a mask)</td>
</tr>
<tr>
<td>MIN (A1, A2 [, A3,...])</td>
<td>E</td>
<td>The minimum value in the set of arguments</td>
</tr>
<tr>
<td>MIN1 (A1, A2 [, A3,...])</td>
<td>E</td>
<td>The minimum value in the set of real arguments (returned as an integer)</td>
</tr>
<tr>
<td>MINEXponent (X)</td>
<td>I</td>
<td>The minimum exponent in the model for the argument</td>
</tr>
<tr>
<td>MINLOC (ARRAY [,DIM] [,MASK])</td>
<td>T</td>
<td>The rank-one array that has the location of the minimum element in the argument array</td>
</tr>
<tr>
<td>MINVAL (ARRAY [,DIM] [,MASK])</td>
<td>T</td>
<td>The minimum value of the elements in the argument array</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MOD (A, P)</td>
<td>E</td>
<td>The remainder of the arguments (has the sign of the first argument)</td>
</tr>
<tr>
<td>MODULO (A, P)</td>
<td>E</td>
<td>The modulo of the arguments (has the sign of the second argument)</td>
</tr>
<tr>
<td>NEAREST (X, S)</td>
<td>E</td>
<td>The nearest different machine-representable number in a given direction</td>
</tr>
<tr>
<td>NINT (A [,KIND])</td>
<td>E</td>
<td>A real value rounded to the nearest integer</td>
</tr>
<tr>
<td>NOT (I)</td>
<td>E</td>
<td>The logical complement of the argument</td>
</tr>
<tr>
<td>NULL ([]MOLD])</td>
<td>T</td>
<td>A disassociated pointer</td>
</tr>
<tr>
<td>OR (I, J)</td>
<td>E</td>
<td>See IOR</td>
</tr>
<tr>
<td>PACK (ARRAY, MASK [,VECTOR])</td>
<td>T</td>
<td>A packed array of rank one (under a mask)</td>
</tr>
<tr>
<td>POPCNT (I)</td>
<td>E</td>
<td>The number of 1 bits in an integer</td>
</tr>
<tr>
<td>POPPAR (I)</td>
<td>E</td>
<td>The parity of an integer</td>
</tr>
<tr>
<td>PRECISION (X)</td>
<td>I</td>
<td>The decimal precision (real or complex) of the argument</td>
</tr>
<tr>
<td>PRESENT (A)</td>
<td>I</td>
<td>.TRUE. if an actual argument has been provided for an optional dummy argument</td>
</tr>
<tr>
<td>PRODUCT (ARRAY [,DIM] [,MASK])</td>
<td>T</td>
<td>The product of the elements of the argument array</td>
</tr>
<tr>
<td>QCMPLX (X, Y)</td>
<td>E</td>
<td>The corresponding COMPLEX(16) value of the argument</td>
</tr>
<tr>
<td>QEXT (A)</td>
<td>E</td>
<td>The corresponding REAL(16) precision value of the argument</td>
</tr>
<tr>
<td>QFLOAT (A)</td>
<td>E</td>
<td>The corresponding REAL(16) precision value of the integer argument</td>
</tr>
<tr>
<td>QREAL (A)</td>
<td>E</td>
<td>The corresponding REAL(16) value of the real part of a COMPLEX(16) argument</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RADIX (X)</td>
<td>I</td>
<td>The base of the model for the argument</td>
</tr>
<tr>
<td>RANGE (X)</td>
<td>I</td>
<td>The decimal exponent range of the model for the argument</td>
</tr>
<tr>
<td>REAL (A [,KIND])</td>
<td>E</td>
<td>The corresponding real value of the argument</td>
</tr>
<tr>
<td>REPEAT (STRING, NCOPIES)</td>
<td>T</td>
<td>The concatenation of zero or more copies of the specified string</td>
</tr>
<tr>
<td>RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])</td>
<td>T</td>
<td>An array that has a different shape than the argument array, but the same elements</td>
</tr>
<tr>
<td>RRSPACING (X)</td>
<td>E</td>
<td>The reciprocal of the relative spacing near the argument</td>
</tr>
<tr>
<td>RSHIFT (I, NEGATIVE_SHIFT)</td>
<td>E</td>
<td>See ISHFT</td>
</tr>
<tr>
<td>SCALE (X, I)</td>
<td>E</td>
<td>The value of the exponent part (of the model for the argument) changed by a specified value</td>
</tr>
<tr>
<td>SCAN (STRING, SET [,BACK])</td>
<td>E</td>
<td>The position of the specified character (or set of characters) within a string</td>
</tr>
<tr>
<td>SELECTED_INT_KIND (R)</td>
<td>T</td>
<td>The integer kind parameter of the argument</td>
</tr>
<tr>
<td>SELECTED_REAL_KIND ([P] [,R])</td>
<td>T</td>
<td>The real kind parameter of the argument; one of the optional arguments must be specified</td>
</tr>
<tr>
<td>SET_EXPONENT (X, I)</td>
<td>E</td>
<td>The value of the exponent part (of the model for the argument) set to a specified value</td>
</tr>
<tr>
<td>SHAPE (SOURCE [,KIND])</td>
<td>I</td>
<td>The shape (rank and extents) of an array or scalar</td>
</tr>
<tr>
<td>SIGN (A, B)</td>
<td>E</td>
<td>A value with the sign transferred from its second argument</td>
</tr>
<tr>
<td>SIN (X)</td>
<td>E</td>
<td>The sine (in radians) of the argument</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SIND (X)</td>
<td>E</td>
<td>The sine (in degrees) of the argument</td>
</tr>
<tr>
<td>SINH (X)</td>
<td>E</td>
<td>The hyperbolic sine of the argument</td>
</tr>
</tbody>
</table>
| SIZE (ARRAY [,DIM]
| [,KIND])         | I    | The size (total number of elements) of the argument array (or one of its dimensions) |
| SNGL (X)          | E    | The corresponding real value of the argument                                 |
| SPACING (X)       | E    | The value of the absolute spacing of model numbers near the argument        |
| SPREAD (SOURCE, DIM, NCOPIES) | T    | A replicated array that has an added dimension                             |
| 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]
| [,KIND])         | I    | The upper bounds of an array (or one of its dimensions)                    |
The following table lists specific functions that have no generic function associated with them:

### Specific Functions with No Generic Association

<table>
<thead>
<tr>
<th>Generic Function</th>
<th>Class</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPROD (X, Y)</td>
<td>E</td>
<td>The double-precision product of two real arguments</td>
</tr>
<tr>
<td>DREAL (A)</td>
<td>E</td>
<td>The corresponding double-precision value of the double-complex argument</td>
</tr>
<tr>
<td>MALLOC (I)</td>
<td>E</td>
<td>The starting address for the block of memory allocated</td>
</tr>
<tr>
<td>MULT_HIGH (I, J)</td>
<td>E</td>
<td>The upper (leftmost) 64 bits of the 128-bit unsigned result</td>
</tr>
<tr>
<td>NUMBER_OF_PROCESSORS ( )</td>
<td>I</td>
<td>The total number of processors (peers) available to the program</td>
</tr>
<tr>
<td>NWORKERS ( )</td>
<td>I</td>
<td>The number of executing processes</td>
</tr>
</tbody>
</table>

---

**Key to Classes**

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

1 Alpha only
2 VMS, U*X
Intrinsic Subroutines

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

### Intrinsic Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_TIME (TIME)</td>
<td>The processor time in seconds</td>
</tr>
<tr>
<td>DATE (BUF)</td>
<td>The ASCII representation of the current date (in dd-mmm-yy form)</td>
</tr>
<tr>
<td>DATE_AND_TIME ([DATE] [,TIME] [,ZONE] [,VALUES])</td>
<td>Date and time information from the real-time clock</td>
</tr>
<tr>
<td>ERRSNS ([IO_ERR] [,SYS_ERR] [,STAT] [,UNIT] [,COND])</td>
<td>Information about the most recently detected error condition</td>
</tr>
</tbody>
</table>
 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 \texttt{IAND}, \texttt{IOR}, \texttt{IEOR}, and \texttt{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 \texttt{ISHFT} and \texttt{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:

<table>
<thead>
<tr>
<th>Binary pattern:</th>
<th>0...010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit position:</td>
<td>n...6543210</td>
</tr>
</tbody>
</table>

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 two's complement notation. For example, the integer -47 is represented by the following:

<table>
<thead>
<tr>
<th>Binary pattern:</th>
<th>1...1010001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit position:</td>
<td>n...6543210</td>
</tr>
</tbody>
</table>

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 x86 processors.

The following example demonstrates **IBSET**, **IBCLR**, and **BTEST**:

```fortran
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, Compaq Fortran provides the following integer data types:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Storage Required (in bytes)</th>
<th>1 Alpha only</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>INTEGER(2)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>INTEGER(4)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>INTEGER(8)</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

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:

```fortran
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 statement
  - ACCEPT statement
- Data transfer output statements:
  - WRITE statement
  - PRINT and TYPE statements
  - REWRITE statement

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

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.
o 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:

o 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.

o 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:

\[
io-keyword \text{ (io-control-list) [io-list]}\]

\[
io-keyword \text{ format [, io-list]}\]

\textit{io-keyword}

Is one of the following: \texttt{ACCEPT}, \texttt{PRINT} (or \texttt{TYPE}), \texttt{READ}, \texttt{REWRITE}, or \texttt{WRITE}.

\textit{io-control-list}

Is one or more of the following input/output (I/O) control specifiers:
**Data Transfer I/O Statements**

**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, the nonkeyword values must appear first. Once a keyword form of a specifier is used, all specifiers to the right must also be keyword forms.

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

- **Unit Specifier**
- **Format Specifier**
- **Namelist Specifier**
- **Record Specifier**
- **I/O Status Specifier**
- **Branch Specifiers**
- **Advance Specifier**
- **Character Count Specifier**

**Unit Specifier**

The unit specifier identifies the I/O unit to be accessed. It takes the following form:

\[
[\text{UNIT=}]\text{io-unit}
\]

\[\text{io-unit}\]
For external files, it identifies a logical unit and is one of the following:

- A scalar integer expression that refers to a specific file, I/O device, or pipe. If necessary, the value is converted to integer data type before use. The integer is in the range 0 through $2^{31}-1$.

Units 5 and 6 are associated with preconnected units. In addition, unit 0 is also associated with a preconnected unit on Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems.

- 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* 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 preconnected units, 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
```

Where

- `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
```

Where

- `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:

\[ \text{KEY}[\text{con}] = \text{val} \]

\text{con}
Is a selection condition keyword specifying how to compare \text{val} with key-field values. The keyword can be any of the following:

**In Ascending-Key Files:**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>The key-field value must be equal to \text{val}. KEYEQ is the same as specifying KEY without the optional \text{con}.</td>
</tr>
<tr>
<td>GE</td>
<td>The key-field value must be greater than or equal to \text{val}.</td>
</tr>
<tr>
<td>GT</td>
<td>The key-field value must be greater than \text{val}.</td>
</tr>
<tr>
<td>NXT</td>
<td>The key-field value must be the next value of the key equal to or greater than \text{val}.</td>
</tr>
<tr>
<td>NXTNE</td>
<td>The key-field value must be the next value of the key strictly greater than \text{val}.</td>
</tr>
</tbody>
</table>

**In Descending-Key Files:**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>The key-field value must be equal to \text{val}. KEYEQ is the same as specifying KEY without the optional \text{con}.</td>
</tr>
<tr>
<td>LE</td>
<td>The key-field value must be less than or equal to \text{val}.</td>
</tr>
<tr>
<td>LT</td>
<td>The key-field value must be less than \text{val}.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>In Ascending-Key Files</th>
<th>In Descending-Key Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYNXT</td>
<td>KEYGE</td>
<td>KEYLE</td>
</tr>
<tr>
<td>KEYNXTNE</td>
<td>KEYGT</td>
<td>KEYLT</td>
</tr>
</tbody>
</table>

The specifiers KEYGE and KEYGT can only be used with ascending-key files, while the specifiers KEYLE and KEYLT can only be used with descending-key files. Any other use of these key specifiers causes a run-time error to occur.

When a program must be able to use either ascending-key or descending-key files, you should use KEYNXT and KEYNXTNE.

**The Selection Process**

To select key-field integer values, the process compares values using the signed integers themselves.

To select key-field character values, the process compares values by using the ASCII collating sequence. The comparative length of val and a key-field value, as well as any specified selection condition, determine the kind of selection that occurs. The selection can be exact, generic, or approximate-generic, as follows:

- Exact selections occur when the expression in val is equal in length to the expression in the key field of the currently accessed record, and the con keyword specifies a unique selection condition.
Generic selections occur when the expression in val is shorter than the expression in the key field of the currently accessed record, and the con keyword specifies a unique selection condition.

The process compares all the characters in val, from left to right, with the same amount of characters in the key field (also from left to right). Remaining key-field characters are ignored.

For example, consider that a record's key field is 10 characters long and the following statement is entered:

```
READ (3, KEYEQ = 'ABCD')
```

In this case, the process can select a record with a key-field value 'ABCDEFGHIJ'.

- An approximate-generic selection occurs when the expression in val is shorter than the expression in the key field, and the con keyword does not specify a unique selection condition.

As with generic selections, the process uses only the leftmost characters in the key field to compare values. It selects the first key field that satisfies the generic selection criterion.

For example, consider that a record's key field is 5 characters long and the following statement is entered:

```
READ (3, KEYGT = 'ABCD')
```

In this case, the process can select the key-field value 'ABCEx' (and not the key-field value 'ABCDA').

If val is longer than the key-field value, no selection is made and a run-time error occurs.

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 StatusSpecifier

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:

\[
\text{IOSTAT}=i\text{-var}
\]

\( i\text{-var} \)
Is a scalar integer variable. When a data transfer statement is executed, \( i\text{-var} \) 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 \texttt{READ} statement; an end-of-record condition occurs only during execution of a nonadvancing \texttt{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:

\[
\begin{align*}
\text{ERR} &= label \\
\text{END} &= label \\
\text{EOR} &= label \\
\end{align*}
\]

*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:

\[ \text{ADVANCE}=c-expr \]

- **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 CountSpecifier

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:

\[ \text{SIZE}=i-var \]

\( i-var \)
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:

```fortran
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:

```fortran
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 \( \text{ARRAY}(J,K) \). The value 721.73 is then assigned to \( \text{ARRAY}(1,3) \). Note that the variables must appear before their use as array subscripts.

Consider the following derived-type definition and structure declaration:

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

... TYPE(EMPLOYEE) :: CONTRACT  ! A structure of type EMPLOYEE
```

The following statements are equivalent:

```plaintext
READ *, CONTRACT

READ *, CONTRACT%ID, CONTRACT%NAME
```

The following shows more examples:

```plaintext
! 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:

```plaintext
! 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:

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

The following example uses a namelist:

```plaintext
! 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)
!
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:

```plaintext
(list, do-var = expr1, expr2 [,expr3])
```

**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:

```latex
WRITE (3,200) (A,B,C, I=1,3)            ! An implied-do list
```

The following example shows nested implied-do lists. Execution of the innermost list is repeated most often:

```latex
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:

```latex
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:

```latex
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:

```fortran
WRITE (6,1111) (I, I=1,20)
```

Integers 1 through 20 are written.

Consider the following:

```fortran
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**

```fortran
```
**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:

```plaintext
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 non-delimited 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:

<table>
<thead>
<tr>
<th>I/O List Item</th>
<th>Value Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>R</td>
<td>6.3</td>
</tr>
<tr>
<td>D</td>
<td>(3.4,4.2)</td>
</tr>
<tr>
<td>E</td>
<td>(3.0,2.0)</td>
</tr>
<tr>
<td>L</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>M</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>J</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
The following example shows list-directed input and output:

```plaintext
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 name list 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:

\[
\&{\text{group-name}} \text{ object} = \text{value} [, \text{object} = \text{value}] \ldots / 
\]

\textit{group-name}

Is the name of the group containing the objects to be given values. The name must have been previously defined in a \textbf{NAMELIST} statement in the scoping unit. The name cannot contain embedded blanks and must be contained within a single record.

\textit{object}

Is the name (or subobject designator) of an entity defined in the \textbf{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.

\textit{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 \textbf{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 **NAMELIST** item is of type default character, and the following is true:

- The character string does not contain a blank, comma (,), slash (/), exclamation point(!), ampersand (&), dollar sign ($), left parenthesis, equal sign (=), percent sign (%), or period (.).
- The character string is not continued across a record boundary.
- The first nonblank character in the string is not an apostrophe or a quotation mark.
- The leading character is not a string of digits followed by an asterisk.

A nondelimited character string is terminated by the first blank, comma, slash, end-of-record, exclamation, ampersand, or dollar sign encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.

If an equal sign, percent sign, or period is encountered while scanning for a nondelimited character string, the string is treated as a variable name (or part of one) and not as a nondelimited character string.

- A null value

A null value is specified by two consecutive value separators (such as ,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

- A repetition of a null value (r*) or a constant (r*constant), where r is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.
Blanks can precede or follow the beginning ampersand (&), follow the group name, precede or follow the equal sign, or precede the terminating slash.

Comments (beginning with ! only) can appear anywhere in namelist input. The comment extends to the end of the source line.

If an entity appears more than once within the input record for a namelist data transfer, the last value is the one that is used.

If there is more than one object = value pair, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Prompting for Namelist Group Information

During execution of a program containing a namelist READ statement, you can specify a question mark character (?) or a question mark character preceded by an equal sign (=?) to get information about the namelist group. The ? or =? must follow one or more blanks.

If specified for a unit capable of both input and output, the ? causes display of the group name and the objects in that group. The =? causes display of the group name, objects within that group, and the current values for those objects (in namelist output form). If specified for another type of unit, the symbols are ignored.

For example, consider the following statements:

```plaintext
NAMELIST /NLIST/ A,B,C
REAL A /1.5/
INTEGER B /2/
CHARACTER*5 C /'ABCDE'/

READ (5,NML=NLIST)
WRITE (6,NML=NLIST)
```
During execution, if a blank followed by ? is entered on a terminal device, the following values are displayed:

\begin{verbatim}
&NLIST
  A
  B
  C
/
\end{verbatim}

If a blank followed by =? is entered, the following values are displayed:

\begin{verbatim}
&NLIST
  A = 1.500000 ,
  B = 2,
  C = ABCDE
/
\end{verbatim}

**Examples**

Suppose the following statements are specified:

```fortran
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:

```fortran
&CONTROL
  TITLE='TESTT002AA',
  INTERVAL=1,
  RESET=.TRUE.,
  START=10.2,
  STOP =14.5
/
```

The following values are assigned to objects in group CONTROL:

<table>
<thead>
<tr>
<th>Namelist Object</th>
<th>Value Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
<td>TESTT002AA</td>
</tr>
<tr>
<td>RESET</td>
<td>T</td>
</tr>
<tr>
<td>START</td>
<td>10.2</td>
</tr>
<tr>
<td>STOP</td>
<td>14.5</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Value Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY_A(1)</td>
<td>1.1</td>
</tr>
<tr>
<td>ARRAY_A(2)</td>
<td>1.2</td>
</tr>
<tr>
<td>ARRAY_A(3)</td>
<td>Unchanged</td>
</tr>
<tr>
<td>ARRAY_A(4)</td>
<td>1.4</td>
</tr>
<tr>
<td>ARRAY_A(5)...ARRAY(20)</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

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:

```fortran
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
 /
```

Nondelimited character strings that are written out by using a NAMELIST write may not be read in as expected by a corresponding NAMELIST read. Consider the following:

```fortran
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/'AAA', 'BBB', 'CCC', 'DDD'/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
WRITE (1, NML=TEST)
END
```

The output file NMLTEST.DAT will contain:

```fortran
&TEST
CHARR   = AAABBBCCCDDD
/
```

If an attempt is then made to read the data in NMLTEST.DAT with a NAMELIST read using nondelimited character strings, as follows:

```fortran
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/4*'   '/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
READ (1, NML=TEST)
PRINT *, 'CHARR read in >', CHARR(1),' >',CHARR(2),' < >',
1       CHARR(3), '< >', CHARR(4), '<'
END
```

The result is the following:

```
CHARR read in >AAA< > < > < > <
```
For More Information:

- 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 \text{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.

\textbf{Examples}

The following example shows a formatted, direct-access \texttt{READ} statement:

\begin{verbatim}
READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
\end{verbatim}

\textbf{Rules for Unformatted Direct-Access READ Statements}

Unformatted, direct-access \texttt{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 \texttt{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.

\textbf{Examples}

The following example shows unformatted, direct-access \texttt{READ} statements:
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'),
```
The \texttt{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 \texttt{ALPHA} and \texttt{BETA}, respectively.

Suppose the following statement is specified:

\begin{verbatim}
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
\end{verbatim}

In this case, the \texttt{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 \texttt{IKEY}.

\section*{Forms and Rules for Internal \texttt{READ} Statements}

Internal \texttt{READ} statements transfer input data from an internal file.

An internal \texttt{READ} statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is not permitted.

An internal \texttt{READ} statement takes the following form:

\begin{verbatim}
READ (iunit, format [,iostat] [,err] [,end]) [io-list]
\end{verbatim}

For more information on syntax, see \texttt{READ} in the \textit{A to Z Reference}.

Formatted, internal \texttt{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 \texttt{READ} statement behaves as if the format begins with a \texttt{BN} edit descriptor. (You can override this behavior by explicitly specifying the \texttt{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.

```fortran
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:

**Formated**

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

**Formated: List-Directed**

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

**Formated: 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
### Default Formats for List-Directed Output

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>I5</td>
</tr>
<tr>
<td>LOGICAL(1)</td>
<td>L2</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td>L2</td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td>L2</td>
</tr>
<tr>
<td>LOGICAL(8)</td>
<td>L2</td>
</tr>
<tr>
<td>INTEGER(1)</td>
<td>I5</td>
</tr>
<tr>
<td>INTEGER(2)</td>
<td>I7</td>
</tr>
<tr>
<td>INTEGER(4)</td>
<td>I12</td>
</tr>
<tr>
<td>INTEGER(8)</td>
<td>I22</td>
</tr>
<tr>
<td>REAL(4)</td>
<td>1PG15.7E2</td>
</tr>
<tr>
<td>REAL(8) T_floating</td>
<td>1PG24.15E3</td>
</tr>
<tr>
<td>REAL(8) D_floating</td>
<td>1PG24.16E2</td>
</tr>
<tr>
<td>REAL(8) G_floating</td>
<td>1PG24.15E3</td>
</tr>
<tr>
<td>REAL(16)</td>
<td>1PG43.33E4</td>
</tr>
<tr>
<td>COMPLEX(4)</td>
<td>('(',1PG14.7E2,',',1PG14.7E2,')')</td>
</tr>
<tr>
<td>COMPLEX(8) T_floating</td>
<td>('(',1PG23.15E3,',',1PG23.15E3,')')</td>
</tr>
<tr>
<td>COMPLEX(8) D_floating</td>
<td>('(',1PG23.16E2,',',1PG23.16E2,')')</td>
</tr>
<tr>
<td>COMPLEX(8) G_floating</td>
<td>('(',1PG23.15E3,',',1PG23.15E3,')')</td>
</tr>
<tr>
<td>COMPLEX(16)</td>
<td>('(',1PG42.33E4,',',1PG42.33E4,')')</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>Aw</td>
</tr>
</tbody>
</table>

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:

```fortran
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 3.40000 3.400000 4
```

The following shows another example:

```fortran
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:

```plaintext
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:

```plaintext
&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:

```plaintext
&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:

```plaintext
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)

The preceding example produces the following output:

&EXAMPLE
INT1 =   11,
INT2 =          12,
INT3 =          14,
LOG1 = T,
LOG2 = T,
LOG3 = T,
REAL1 =   24.00000    ,
REAL2 =   28.0000000000000     ,
Z1    = (38.0000,0.0000000E+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 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):

```fortran
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:

```fortran
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:

- Format specifications
- Data edit descriptors
- Control edit descriptors
- Character string edit descriptors
- Nested and group repeat specifications
- Variable Format Expressions
- Printing of formatted records
- Interaction between FORMAT statements and I/O lists

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
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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A[w]</td>
<td>Transfers character or Hollerith values.</td>
</tr>
<tr>
<td>B</td>
<td>Bw[.m]</td>
<td>Transfers binary values.</td>
</tr>
<tr>
<td>BN</td>
<td>BN</td>
<td>Ignores embedded and trailing blanks in a numeric input field.</td>
</tr>
<tr>
<td>BZ</td>
<td>BZ</td>
<td>Treats embedded and trailing blanks in a numeric input field as zeros.</td>
</tr>
<tr>
<td>D</td>
<td>Dw.d</td>
<td>Transfers real values with D exponents.</td>
</tr>
<tr>
<td>E</td>
<td>Ew.d[Ee]</td>
<td>Transfers real values with E exponents.</td>
</tr>
<tr>
<td>EN</td>
<td>ENw.d[Ee]</td>
<td>Transfers real values with engineering notation.</td>
</tr>
<tr>
<td>ES</td>
<td>ESw.d[Ee]</td>
<td>Transfers real values with scientific notation.</td>
</tr>
<tr>
<td>F</td>
<td>Fw.d</td>
<td>Transfers real values with no exponent.</td>
</tr>
<tr>
<td>Code</td>
<td>Example</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>G</td>
<td>Gw.d [Ee]</td>
<td>Transfers values of all intrinsic types.</td>
</tr>
<tr>
<td>H</td>
<td>nhch [ch...]</td>
<td>Transfers characters following the H edit descriptor to an output record.</td>
</tr>
<tr>
<td>I</td>
<td>Iw[.m]</td>
<td>Transfers decimal integer values.</td>
</tr>
<tr>
<td>L</td>
<td>Lw</td>
<td>Transfers logical values: on input, transfers characters; on output, transfers T or F.</td>
</tr>
<tr>
<td>O</td>
<td>Ow[.m]</td>
<td>Transfers octal values.</td>
</tr>
<tr>
<td>P</td>
<td>kp</td>
<td>Interprets certain real numbers with a specified scale factor.</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>Returns the number of characters remaining in an input record.</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>Writes optional plus sign (+) into numeric output fields.</td>
</tr>
<tr>
<td>SS</td>
<td>SS</td>
<td>Suppresses optional plus sign (+) in numeric output fields.</td>
</tr>
<tr>
<td>T</td>
<td>Tn</td>
<td>Tabs to specified position.</td>
</tr>
<tr>
<td>TL</td>
<td>TLn</td>
<td>Tabs left the specified number of positions.</td>
</tr>
<tr>
<td>TR</td>
<td>TRn</td>
<td>Tabs right the specified number of positions.</td>
</tr>
<tr>
<td>X</td>
<td>nX</td>
<td>Skips the specified number of positions.</td>
</tr>
<tr>
<td>Z</td>
<td>Zw[.m]</td>
<td>Transfers hexadecimal values.</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>Suppresses trailing carriage return during interactive I/O.</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>Terminates format control if there are no more items in the I/O list.</td>
</tr>
<tr>
<td>[r]/</td>
<td>[r]/</td>
<td>Terminates the current record and moves to the next record.</td>
</tr>
</tbody>
</table>
| \\
| \\
| Continues the same record; same as $. |
I/O Formatting

Character Format Specifications

In data transfer I/O statements, a format specifier ([FMT=]format) can be a character expression that is a character array, character array element, or character constant. This type of format is also called a run-time format because it can be constructed or altered during program execution.

The expression must evaluate to a character string whose leading part is a valid format specification (including the enclosing parentheses).

If the expression is a character array element, the format specification must be contained entirely within that element.

If the expression is a character array, the format specification can continue past the first element into subsequent consecutive elements.

If the expression is a character constant delimited by apostrophes, use two consecutive apostrophes (''') to represent an apostrophe character in the format specification; for example:

```
PRINT '("NUM can't be a real number")'
```

Similarly, if the expression is a character constant delimited by quotation marks, use two consecutive quotation marks (""""""") to represent a quotation mark character in the format specification.

To avoid using consecutive apostrophes or quotation marks, you can put the character constant in an I/O list instead of a format specification, as follows:

```
PRINT "(A)", "NUM can't be a real number"
```

The following shows another character format specification:

```
WRITE (6, '(I12, I4, I12)') I, J, K
```

In the following example, the format specification changes with each iteration of the DO loop:

```
SUBROUTINE PRINT(TABLE)
REAL TABLE(10,5)
CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG, FMED, FSML
DATA FORCHR(0),RPAR /'(',')'/
DATA FBIG,FMED,FSML /'F8.2,,F9.4,,F9.6,/'
DO I=1,10
```
DO J=1,5
  IF (TABLE(I,J) .GE. 100.) THEN
    FORCHR(J) = FBIG
  ELSE IF (TABLE(I,J) .GT. 0.1) THEN
    FORCHR(J) = FMED
  ELSE
    FORCHR(J) = FSML
  END IF
END DO
FORCHR(5)(5:5) = RPAR
WRITE (6, FORCHR) (TABLE(I,J), J=1,5)
END DO
END

The **DATA** statement assigns a left parenthesis to character array element FORCHR(0), and (for later use) a right parenthesis and three F edit descriptors to character variables.

Next, the proper F edit descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of array TABLE.

A right parenthesis is added to the format specification just before the **WRITE** statement uses it.

---

**Note:** Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a **READ** statement to read data into the format itself, that data is not copied back into the original array, and the array is unavailable for subsequent use as a run-time format specification.

---

**Examples**

The following example shows a format specification:

```fortran
WRITE (*, 9000) int1, real1(3), char1
9000   FORMAT (I5, 3F4.5, A16)
!  I5, 3F4.5, 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.

```fortran
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:

```fortran
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:

```fortran
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:

```fortran
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:

```fortran
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](#)
- [Integer Editing](#)
- [Real and Complex 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). If omitted, the system applies default values (see Default Widths for Data Edit Descriptors). The range of w is 1 through 2147483647 (2**31-1) on Alpha processors; 1 through 32767 (2**15-1) on x86 processors. For I, B, O, Z, and F, the range can start at zero.

m  
Is the minimum number of digits that must be in the field (including leading zeros). The range of m is 0 through 32767 (2**15-1) on Alpha processors; 0 through 255 (2**8-1) on x86 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 x86 processors.

The number of significant digits is affected if a scale factor is specified for the data edit descriptor.

E  
Identifies an exponent field.
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 x86 processors.

Rules and Behavior

Fortran 95/90 (and the previous standard) allows the field width to be omitted only for the A descriptor. However, Compaq 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: \( Iw[m], Bw[m], Ow[m], \) and \( Zw[m] \)
- Real and complex: \( Fw.d, Ew.d[Ee], ENw.d[Ee], ESw.d[Ee], Dw.d, \) and \( Gw.d[Ee] \)
- Logical: \( Lw \)
- Character: \( A[w] \)

The d must be specified with F, E, D, and G field descriptors even if d is zero. The decimal point is also required. You must specify both w and d, or omit them both.

A repeat specification can simplify formatting. For example, the following two statements are equivalent:

```
20 FORMAT (E12.4, E12.4, E12.4, I5, I5, I5, I5)
20 FORMAT (3E12.4, 4I5)
```

Examples

```fortran
! 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
```
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:

<table>
<thead>
<tr>
<th>Type of Unit or File</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>An explicitly <strong>OPENed</strong>ed unit</td>
<td><code>BLANK='NULL'</code></td>
</tr>
<tr>
<td>An internal file</td>
<td><code>BLANK='NULL'</code></td>
</tr>
<tr>
<td>A preconnected file(^1)</td>
<td><code>BLANK='NULL'</code></td>
</tr>
</tbody>
</table>

\(^1\) 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:

$Iw[.m]

The value of $m$ (the minimum number of digits in the constant) must not exceed the value of $w$ (the field width). The $m$ has no effect on input, only output.

The specified I/O list item must be of type integer or logical.
The **G** edit descriptor can be used to edit integer data; it follows the same rules as **Iw**.

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I4</td>
<td>2788</td>
<td>2788</td>
</tr>
<tr>
<td>I3</td>
<td>-26</td>
<td>-26</td>
</tr>
<tr>
<td>I9</td>
<td>^^^^^^312</td>
<td>312</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>I3</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>I4</td>
<td>-284</td>
<td>-284</td>
</tr>
<tr>
<td>I4</td>
<td>0</td>
<td>^^^0</td>
</tr>
<tr>
<td>I5</td>
<td>174</td>
<td>^174</td>
</tr>
<tr>
<td>I2</td>
<td>3244</td>
<td>**</td>
</tr>
<tr>
<td>I3</td>
<td>-473</td>
<td>***</td>
</tr>
<tr>
<td>I7</td>
<td>29.812</td>
<td>An error; the decimal point is invalid</td>
</tr>
<tr>
<td>I4.0</td>
<td>0</td>
<td>^^^^</td>
</tr>
<tr>
<td>I4.2</td>
<td>1</td>
<td>^01</td>
</tr>
<tr>
<td>I4.4</td>
<td>1</td>
<td>0001</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B4</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>B1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>B4</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>B2</td>
<td>0</td>
<td>^0</td>
</tr>
</tbody>
</table>

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:

\[ \text{Ow}[.m] \]

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>O5</td>
<td>32767</td>
<td>32767</td>
</tr>
<tr>
<td>O4</td>
<td>16234</td>
<td>1623</td>
</tr>
<tr>
<td>O3</td>
<td>97^</td>
<td>An error; the 9 is invalid in octal notation</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>O6</td>
<td>32767</td>
<td>^77777</td>
</tr>
<tr>
<td>O12</td>
<td>-32767</td>
<td>^37777700001</td>
</tr>
<tr>
<td>O2</td>
<td>14261</td>
<td>**</td>
</tr>
<tr>
<td>O4</td>
<td>27</td>
<td>^33</td>
</tr>
<tr>
<td>O5</td>
<td>10.5</td>
<td>41050</td>
</tr>
<tr>
<td>O4.2</td>
<td>7</td>
<td>^07</td>
</tr>
<tr>
<td>O4.4</td>
<td>7</td>
<td>0007</td>
</tr>
</tbody>
</table>

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 through 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A94</td>
<td>A94</td>
</tr>
<tr>
<td>25</td>
<td>A23DEF</td>
<td>A23DE</td>
</tr>
<tr>
<td>25</td>
<td>95.AF2</td>
<td>An error; the decimal point is invalid</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>32767</td>
<td>7FFF</td>
</tr>
<tr>
<td>29</td>
<td>-32767</td>
<td>^FFFF8001</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>24</td>
<td>-10.5</td>
<td>****</td>
</tr>
<tr>
<td>23.3</td>
<td>2708</td>
<td>A94</td>
</tr>
<tr>
<td>26.4</td>
<td>2708</td>
<td>^^0A94</td>
</tr>
</tbody>
</table>

**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:

\[ F_{w.d} \]

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:
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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8.5</td>
<td>2.3547188</td>
<td>^2.35472</td>
</tr>
<tr>
<td>F9.3</td>
<td>8789.7361</td>
<td>^8789.736</td>
</tr>
<tr>
<td>F2.1</td>
<td>51.44</td>
<td>**</td>
</tr>
<tr>
<td>F10.4</td>
<td>-23.24352</td>
<td>^^-23.2435</td>
</tr>
<tr>
<td>F5.2</td>
<td>325.013</td>
<td>*****</td>
</tr>
<tr>
<td>F5.2</td>
<td>-.2</td>
<td>-.20</td>
</tr>
</tbody>
</table>

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:

\[
E^{w.d}[Ee] \\
D^{w.d}
\]

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9.3</td>
<td>734.432E3</td>
<td>734320</td>
</tr>
<tr>
<td>E12.4</td>
<td>^1022.43E</td>
<td>1022.43E-6</td>
</tr>
<tr>
<td>E15.3</td>
<td>52.3759663^^^^^</td>
<td>52.3759663</td>
</tr>
<tr>
<td>E12.5</td>
<td>210.5271D+10(^1)</td>
<td>210.5271E10</td>
</tr>
<tr>
<td>BZ,D10.2</td>
<td>12345^^^^^</td>
<td>12345000.0D0</td>
</tr>
<tr>
<td>D10.2</td>
<td>^123.45^^</td>
<td>123.45D0</td>
</tr>
<tr>
<td>D15.3</td>
<td>367.4981763D+04</td>
<td>3.674981763D+06</td>
</tr>
</tbody>
</table>

\(^1\) 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:

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Positive Form of Exponent</th>
<th>Negative Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ew.d</td>
<td>(</td>
<td>\exp</td>
<td>\leq 99)</td>
</tr>
<tr>
<td></td>
<td>(99 &lt;</td>
<td>\exp</td>
<td>\leq 999)</td>
</tr>
<tr>
<td>Ew.dEe</td>
<td>(</td>
<td>\exp</td>
<td>\leq 10^e - 1)</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>E11.2</td>
<td>475867.222</td>
<td>^^^0.48E+06</td>
</tr>
<tr>
<td>E11.5</td>
<td>475867.222</td>
<td>0.47587E+06</td>
</tr>
<tr>
<td>E12.3</td>
<td>0.00069</td>
<td>^^^0.690E</td>
</tr>
<tr>
<td>E10.3</td>
<td>-0.5555</td>
<td>-0.556E+00</td>
</tr>
<tr>
<td>E5.3</td>
<td>56.12</td>
<td>*****</td>
</tr>
<tr>
<td>E14.5E4</td>
<td>-1.001</td>
<td>-0.10010E+0001</td>
</tr>
<tr>
<td>E13.3E6</td>
<td>0.000123</td>
<td>0.123E-000003</td>
</tr>
<tr>
<td>D14.3</td>
<td>0.0363</td>
<td>^^^^0.363D-01</td>
</tr>
<tr>
<td>D23.12</td>
<td>5413.87625793</td>
<td>^^^^0.541387625793D+04</td>
</tr>
<tr>
<td>D9.6</td>
<td>1.2</td>
<td>*********</td>
</tr>
</tbody>
</table>

**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:

\[ \text{EN} w.d[Ee] \]

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN11.3</td>
<td>^5.321E+00</td>
<td>5.32100</td>
</tr>
<tr>
<td>EN11.3</td>
<td>-600.00E-03</td>
<td>-.60000</td>
</tr>
<tr>
<td>EN12.3</td>
<td>^3.150E-03</td>
<td>.00315</td>
</tr>
<tr>
<td>EN12.3</td>
<td>^3.829E+03</td>
<td>3829.0</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Positive Form of Exponent</th>
<th>Negative Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENw.d</td>
<td></td>
<td></td>
<td>E+nn</td>
</tr>
<tr>
<td></td>
<td>99 &lt;</td>
<td>exp</td>
<td>&lt;= 999</td>
</tr>
<tr>
<td>ENw.dEe</td>
<td></td>
<td></td>
<td>E+n_1n_2...n_e</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN11.2</td>
<td>475867.222</td>
<td>^475.87E+03</td>
</tr>
<tr>
<td>EN11.5</td>
<td>475867.222</td>
<td>**************</td>
</tr>
<tr>
<td>EN12.3</td>
<td>0.00069</td>
<td>^690.000E-06</td>
</tr>
<tr>
<td>EN10.3</td>
<td>-0.5555</td>
<td>**********</td>
</tr>
<tr>
<td>EN11.2</td>
<td>0.0</td>
<td>^000.00E-03</td>
</tr>
</tbody>
</table>

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:

\[
\text{ES} w.d[Ee]
\]

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES11.3</td>
<td>^^5.321E+00</td>
<td>5.32100</td>
</tr>
<tr>
<td>ES11.3</td>
<td>~6.000E-03</td>
<td>-.60000</td>
</tr>
<tr>
<td>ES12.3</td>
<td>^^^3.150E-03</td>
<td>-.00315</td>
</tr>
<tr>
<td>ES12.3</td>
<td>^^^3.829E+03</td>
<td>3829.0</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Absolute Value of Exponent</th>
<th>Positive Form of Exponent</th>
<th>Negative Form of Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESw.d</td>
<td>(</td>
<td>\exp</td>
<td>\leq 99)</td>
</tr>
<tr>
<td></td>
<td>99 &lt; (</td>
<td>\exp</td>
<td>\leq 999)</td>
</tr>
<tr>
<td>ESw.dEe</td>
<td>(</td>
<td>\exp</td>
<td>\leq 10^e - 1)</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES11.2</td>
<td>473214.356</td>
<td>^^^4.73E+05</td>
</tr>
<tr>
<td>ES11.5</td>
<td>473214.356</td>
<td>4.73214E+05</td>
</tr>
<tr>
<td>ES12.3</td>
<td>0.00069</td>
<td>^^^6.900E-04</td>
</tr>
<tr>
<td>ES10.3</td>
<td>-0.5555</td>
<td>-5.555E-01</td>
</tr>
<tr>
<td>ES11.2</td>
<td>0.0</td>
<td>^0.000E+00</td>
</tr>
</tbody>
</table>

For More Information:
- See Forms for data edit descriptors.
- See General rules for numeric 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 \( Iw \), \( Lw \), and \( Aw \), 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>
<thead>
<tr>
<th>Data Magnitude</th>
<th>Effective Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 &lt; m &lt; 0.1 - 0.5 \times 10^{-d-1} )</td>
<td>( Ew.d[Ee] )</td>
</tr>
<tr>
<td>( m = 0 )</td>
<td>( F(w - n).(d -1), n('b') )</td>
</tr>
<tr>
<td>( 0.1 - 0.5 \times 10^{-d-1} \leq m &lt; 1 - 0.5 \times 10^{-d} )</td>
<td>( F(w - n).d, n('b') )</td>
</tr>
<tr>
<td>( 1 - 0.5 \times 10^{-d} \leq m &lt; 10 - 0.5 \times 10^{-d+1} )</td>
<td>( F(w - n).(d -1), n('b') )</td>
</tr>
<tr>
<td>( 10 - 0.5 \times 10^{-d+1} \leq m &lt; 100 - 0.5 \times 10^{-d+2} )</td>
<td>( F(w - n).(d -2), n('b') )</td>
</tr>
</tbody>
</table>
The 'b' is a blank following the numeric data representation. For \texttt{Gw.d}, \( n('b') \) is 4 blanks. For \texttt{Gw.dEe}, \( n('b') \) is \( e+2 \) blanks.

The \( w \) should be greater than or equal to \( d+7 \) to allow for the following:
- A sign (optional if the value is positive and descriptor \texttt{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 \texttt{G} edit descriptor and compares it to output using equivalent \texttt{F} editing (the symbol ^ represents a nonprinting blank character):

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Output with G</th>
<th>Format</th>
<th>Output with F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01234567</td>
<td>G13.6</td>
<td>^0.123457E-01</td>
<td>F13.6</td>
<td>^^^^^0.012346</td>
</tr>
<tr>
<td>-0.12345678</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^-0.123457</td>
</tr>
<tr>
<td>1.23456789</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^1.234568</td>
</tr>
<tr>
<td>12.34567890</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^12.345679</td>
</tr>
<tr>
<td>123.45678901</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^123.45679</td>
</tr>
<tr>
<td>-1234.56789012</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^-1234.567890</td>
</tr>
<tr>
<td>1234.567890123</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^123456.789012</td>
</tr>
<tr>
<td>12345.678901234</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>123456.789012</td>
</tr>
<tr>
<td>-123456.789012345</td>
<td>G13.6</td>
<td>^0.123457E+00</td>
<td>F13.6</td>
<td>^^^^^123456.789012</td>
</tr>
</tbody>
</table>

For More Information:
- See Forms for data edit descriptors.
- See General rules for numeric editing.
- See the \texttt{I} data edit descriptor.
- See the \texttt{L} data edit descriptor.
- See the \texttt{A} data edit descriptor.
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: \texttt{Fw.d}, \texttt{Ew.d[Ee]}, \texttt{Dw.d}, \texttt{ENw.d[Ee]}, \texttt{ESw.d[Ee]}, or \texttt{Gw.d[Ee]}.

Rules for Input Processing

On input, the two successive fields are read and assigned to the corresponding complex I/O list item as its real and imaginary part, respectively.

The following shows input using complex editing:

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{F8.5,F8.5}</td>
<td>1234567812345.67</td>
<td>123.45678, 12345.67</td>
</tr>
<tr>
<td>\texttt{E9.1,F9.3}</td>
<td>734.432E8123456789</td>
<td>734.432E8, 123456.789</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{2F8.5}</td>
<td>2.3547188, 3.456732</td>
<td>^2.35472 ^3.45673</td>
</tr>
<tr>
<td>\texttt{E9.2,'^,^',E5.3}</td>
<td>47587.222, 56.123</td>
<td>^0.48E+06 ^,****</td>
</tr>
</tbody>
</table>

For More Information:

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

Logical Editing (L)

The \texttt{L} data edit descriptor transfers logical values. It takes the following form:

\texttt{Lw}

The specified I/O list item must be of type logical or integer.

The \texttt{G} edit descriptor can be used to edit logical data; it follows the same rules.
as \texttt{Lw}.

**Rules for Input Processing**

On input, the \texttt{L} data edit descriptor transfers \textit{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:

- \texttt{.TRUE.} is assigned if the first nonblank character is \texttt{T}, \texttt{.T}, \texttt{.t}, or \texttt{t}. The logical constant \texttt{.TRUE.} is an acceptable input form.
- \texttt{.FALSE.} is assigned if the first nonblank character is \texttt{F}, \texttt{.F}, \texttt{.f}, or \texttt{f}, or the entire field is filled with blanks. The logical constant \texttt{.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 \texttt{L} data edit descriptor transfers the following to an external field that is \textit{w} characters long: \texttt{w} - 1 blanks, followed by a \texttt{T} or \texttt{F} (if the value is \texttt{.TRUE.} or \texttt{.FALSE.}, respectively).

The following shows output using the \texttt{L} edit descriptor (the symbol ^ represents a nonprinting blank character):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>L5</td>
<td>.TRUE.</td>
<td>^^^^T</td>
</tr>
<tr>
<td>L1</td>
<td>.FALSE.</td>
<td>F</td>
</tr>
</tbody>
</table>

For More Information:

See [Forms for data edit descriptors](#).

**Character Editing (A)**

The \texttt{A} data edit descriptor transfers character or Hollerith values. It takes the following form:

\[
\texttt{A[w]}
\]

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 \texttt{G} edit descriptor can be used to edit character data; it follows the same rules as \texttt{Aw}.
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:

<table>
<thead>
<tr>
<th>I/O List Element</th>
<th>Maximum Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL(1) or LOGICAL*1</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL(2) or LOGICAL*2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL(4) or LOGICAL*4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL(8) or LOGICAL*8</td>
<td>( 8^1 )</td>
</tr>
<tr>
<td>INTEGER(1) or INTEGER*1</td>
<td>1</td>
</tr>
<tr>
<td>INTEGER(2) or INTEGER*2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER(4) or INTEGER*4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER(8) or INTEGER*8</td>
<td>( 8^1 )</td>
</tr>
<tr>
<td>REAL(4) or REAL*4</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>REAL(8) or REAL*8</td>
<td>8</td>
</tr>
<tr>
<td>REAL(16) or REAL*16</td>
<td>( 16^2 )</td>
</tr>
<tr>
<td>COMPLEX(4) or COMPLEX*8</td>
<td>( 8^3 )</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>#</td>
<td>CHARACTER (LEN=1)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>E^#</td>
<td>CHARACTER (LEN=3)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>PAGE^#</td>
<td>CHARACTER (LEN=6)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>PAGE^#^^</td>
<td>CHARACTER (LEN=8)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>#</td>
<td>LOGICAL (1)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>^#</td>
<td>INTEGER (2)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>GE^#</td>
<td>REAL (4)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE^#</td>
<td>PAGE^#^^</td>
<td>REAL (8)</td>
</tr>
</tbody>
</table>

**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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>OHMS</td>
<td>^OHMS</td>
</tr>
<tr>
<td>A5</td>
<td>VOLTS</td>
<td>VOLTS</td>
</tr>
<tr>
<td>A5</td>
<td>AMPERES</td>
<td>AMPER</td>
</tr>
</tbody>
</table>

**For More Information:**

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.)
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

<table>
<thead>
<tr>
<th>Edit Descriptor</th>
<th>Data Type of I/O List Item</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I, B, O, Z, G )</td>
<td>BYTE</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>INTEGER(1), LOGICAL(1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2), LOGICAL(2)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>INTEGER(4), LOGICAL(4)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>INTEGER(8(^1)), LOGICAL(8(^1))</td>
<td>23</td>
</tr>
<tr>
<td>( O, Z )</td>
<td>REAL(4)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>REAL(8)</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>REAL(16(^2))</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>CHARACTER*len</td>
<td>MAX(7, 3*len)</td>
</tr>
<tr>
<td>( L, G )</td>
<td>LOGICAL(1), LOGICAL(2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>LOGICAL(4), LOGICAL(8(^1))</td>
<td></td>
</tr>
<tr>
<td>( F, E, EN, ES, G, D )</td>
<td>REAL(4), COMPLEX(4)</td>
<td>15 ( d: 7 ) ( e: 2 )</td>
</tr>
<tr>
<td></td>
<td>REAL(8), COMPLEX(8)</td>
<td>25 ( d: 16 ) ( e: 2 )</td>
</tr>
<tr>
<td></td>
<td>REAL(16(^2)), COMPLEX(16(^2))</td>
<td>42 ( d: 33 ) ( e: 3 )</td>
</tr>
<tr>
<td>( A (^3), G )</td>
<td>LOGICAL(1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LOGICAL(2), INTEGER(2)</td>
<td>2</td>
</tr>
</tbody>
</table>
Terminating Short Fields of Input Data

On input, an edit descriptor such as \texttt{Fw.d} specifies that \( w \) characters (the field width) are to be read from the external field.

If the field contains fewer than \( w \) characters, the input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data.

Padding Short Fields

You can use the \texttt{OPEN} statement specifier \texttt{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:

\begin{verbatim}
READ (2, '(I5)') J
\end{verbatim}

If 3 is input for \( J \), the value of \( J \) will be 30000 or 3 depending on which default behavior is in effect (\texttt{BLANK='NULL'} or \texttt{BLANK='ZERO'}). This can give unexpected results.

To ensure that the desired behavior is in effect, explicitly specify the \texttt{BN} or \texttt{BZ} edit descriptor. For example, the following ensures that blanks are interpreted as blanks (and not as zeros):

\begin{verbatim}
READ (2, '(BN, I5)') J
\end{verbatim}

Using Commas to Separate Input Data

\begin{verbatim}
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL(4), INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL(8), INTEGER(8)</td>
<td>8</td>
</tr>
<tr>
<td>REAL(4), COMPLEX(4)</td>
<td>4</td>
</tr>
<tr>
<td>REAL(8), COMPLEX(8)</td>
<td>8</td>
</tr>
<tr>
<td>REAL(16), COMPLEX(16)</td>
<td>16</td>
</tr>
<tr>
<td>CHARACTER*len</td>
<td>len</td>
</tr>
</tbody>
</table>
\end{verbatim}

\(^1\) Alpha only
\(^2\) VMS, U*X
\(^3\) The default is the actual length of the corresponding I/O list item.
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:

```plaintext
READ (5,100) I,J,A,B
100 FORMAT (2I6,2F10.2)
```

Suppose a record containing the following values is read:

1, -2, 1.0, 35

The following assignments occur:

- $I = 1$
- $J = -2$
- $A = 1.0$
- $B = 0.35$

A comma can only terminate fields less than $w$ characters long. If a comma follows a field of $w$ or more characters, the comma is considered part of the next field.

A null (zero-length) field is designated by two successive commas, or by a comma after a field of $w$ characters. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.0D0, 0.0Q0, or .FALSE..

**For More Information:**

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 (:)

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: \( T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, \backslash, $, \) 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 x86 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: \( :, /, \backslash, $, \) 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:

```plaintext
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:

```
Xn
```

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 \( \mathbf{X} \) edit descriptor does not output any characters when it appears at the end of a format specification; for example:

\[
\text{WRITE (6,99) K} \\
99 \quad \text{FORMAT ('^K=',I6,5X)}
\]

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 \( \mathbf{S} \), \( \mathbf{SP} \), and \( \mathbf{SS} \) 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 \( \mathbf{I} \), \( \mathbf{F} \), \( \mathbf{E} \), \( \mathbf{EN} \), \( \mathbf{ES} \), \( \mathbf{D} \), and \( \mathbf{G} \) 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 \( \mathbf{SP} \) 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:

\[
\mathbf{SP}
\]
**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:

```
kP
```

The **k** is a signed (sign is optional if positive), integer literal constant specifying the number of positions, to the left or right, that the decimal point is to move (the scale factor). The range of **k** is -128 to 127.

At the beginning of a formatted I/O statement, the value of the scale factor is zero. If a scale editing descriptor is specified, the scale factor is set to the new value, which affects all subsequent real edit descriptors until another scale editing descriptor occurs.

To reinstate a scale factor of zero, you must explicitly specify **0P**.

Format reversion does not affect the scale factor. (For more information on format reversion, see [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 **2P** scale factor multiplies an input value by .01, moving the decimal point two places to the left. A **-2P** scale factor multiplies an input value by 100, moving the decimal point two places to the
right.

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3PE10.5</td>
<td>^^^37.614^</td>
<td>.037614</td>
</tr>
<tr>
<td>3PE10.5</td>
<td>^^37.614E2</td>
<td>3761.4</td>
</tr>
<tr>
<td>-3PE10.5</td>
<td>^^^^37.614</td>
<td>37614.0</td>
</tr>
</tbody>
</table>

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 \( \text{EN} \) and \( \text{ES} \) editing, the scale factor has no effect.

The following shows output using the \( P \) edit descriptor (the symbol ^ represents a nonprinting blank character):

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1PE12.3</td>
<td>-270.139</td>
<td>^^-2.701E+02</td>
</tr>
<tr>
<td>1P,E12.2</td>
<td>-270.139</td>
<td>^^^-2.70E+02</td>
</tr>
<tr>
<td>-1PE12.2</td>
<td>-270.139</td>
<td>^^^-0.03E+04</td>
</tr>
</tbody>
</table>

**Examples**

The following shows a **FORMAT** statement containing a scale factor:

```plaintext
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:

```plaintext
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  12340000  12340000
12.34     12.34     12.34     12.34     12.34     12.34
12.34e0   12.34e0   12.34e0   12.34e0   12.34e0   12.34e0
12.34e3   12.34e3   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  1.234  1.234  1234.000  12.340  1.234  1.234  1234.000
12.340  1.234  1.234  1234.000  12.340  1.234  1.234  1234.000
12340.000  12340.000  12340.000  12340.000
```

The next code shows scale-factor editing when writing:

```plaintext
a = 12.34
WRITE (*, 100) a, a, a, a, a
```
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 x86 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:

```plaintext
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
```
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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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:

```plaintext
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:

```plaintext
CHARACTER(20) MYNAME
WRITE (*,9000)
9000 FORMAT ('Please 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:

```plaintext
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:

```plaintext
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:

```fortran
! 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:

```plaintext
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 a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Compaq Fortran fully supports features deleted in Fortran 95.

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

```
*nH*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 x86 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

```plaintext
! 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.
```
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 (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 or in character format specifications.

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:

```fortran
 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:

```fortran
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
```
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(I, 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:

<table>
<thead>
<tr>
<th>Control Characters for Printing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Overprinting</td>
<td>Outputs the record (at the current position in the current line) and a carriage return.</td>
</tr>
<tr>
<td>-</td>
<td>One line feed</td>
<td>Outputs the record (at the beginning of the following line) and a carriage return.</td>
</tr>
<tr>
<td>0</td>
<td>Two line feeds</td>
<td>Outputs the record (after skipping a line) and a carriage return.</td>
</tr>
<tr>
<td>1</td>
<td>Next page</td>
<td>Outputs the record (at the beginning of a new page) and a carriage return.</td>
</tr>
<tr>
<td>$</td>
<td>Prompting</td>
<td>Outputs the record (at the beginning of the following line), but no carriage return.</td>
</tr>
<tr>
<td>ASCII NUL</td>
<td>Overprinting with no advance</td>
<td>Outputs the record (at the current position in the current line), but no carriage return.</td>
</tr>
</tbody>
</table>

1 Specify as CHAR(0).
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
```
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)) ) 4

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

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

READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2) 7
300   FORMAT ( (I3, X, 5(I4,X)) ) 8

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

400   FORMAT ( I3, X, 5(I4,X) )

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

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
```
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.

This statement produces the following output:

\[
\begin{array}{c}
B \text{ is} & 4 & 5 \\
\text{is} & A & \\
401 & 402 & 403 & 404 & 405 \\
501 & 502 & 503 & 504 & 505
\end{array}
\]

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.

This statement produces the following output:

\[
\begin{array}{c}
B \text{ is} & 6 & 7 \\
\text{is} & A & \\
601 & 602 & 603 & 604 & 605 \\
701 & 702 & 703 & 704 & 705
\end{array}
\]

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.

This statement produces the following output:

\[
\begin{array}{c}
B \text{ is} & 8 & 90 \\
\text{is} & A & \\
801 & 802 & 803 & 804 & 805 \\
9010 & 9020 & 9030 & 9040 & 100
\end{array}
\]

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, W9*, U*X)**

The following are file connection, inquiry, and positioning I/O statements on Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux 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

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Values</th>
<th>Description</th>
<th>Used with:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS=access</td>
<td>'SEQUENTIAL', 'DIRECT', or 'APPEND'</td>
<td>Specifies the method of file access.</td>
<td><strong>INQUIRE</strong> <strong>OPEN</strong></td>
</tr>
<tr>
<td>ACTION=permission</td>
<td>'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')</td>
<td>Specifies file I/O mode.</td>
<td><strong>INQUIRE</strong> <strong>OPEN</strong></td>
</tr>
<tr>
<td>ADVANCE=c-expr</td>
<td>'NO' or 'YES' (default is 'YES')</td>
<td>Specifies formatted sequential data input as advancing, or non-advancing.</td>
<td><strong>READ</strong></td>
</tr>
<tr>
<td>ASSOCIATEVARIABLE=var</td>
<td>Integer variable</td>
<td>Specifies a variable to be updated to reflect the record number of the next sequential record in the file.</td>
<td><strong>OPEN</strong></td>
</tr>
<tr>
<td>BINARY=bin</td>
<td>'NO' or 'YES'</td>
<td>Returns whether file format is binary.</td>
<td><strong>INQUIRE</strong></td>
</tr>
<tr>
<td>BLANK=blank_control</td>
<td>'NULL' or 'ZERO' (default is 'NULL')</td>
<td>Specifies whether blanks are ignored in numeric fields or interpreted as zeros.</td>
<td><strong>INQUIRE</strong> <strong>OPEN</strong></td>
</tr>
<tr>
<td>BLOCKSIZE=blocksize</td>
<td>Positive integer variable or expression</td>
<td>Specifies or returns the internal buffer size used in I/O.</td>
<td><strong>INQUIRE</strong> <strong>OPEN</strong></td>
</tr>
<tr>
<td>BUFFERCOUNT=$bc</td>
<td>Numeric expression</td>
<td>Specifies the number of buffers to be associated with the unit for multibuffered I/O.</td>
<td>OPEN</td>
</tr>
<tr>
<td>BUFFERED=$bf</td>
<td>'YES' or 'NO' (default is 'NO')</td>
<td>Specifies runtime library behavior following WRITE operations.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>CARRIAGECONTROL=$control</td>
<td>'FORTRAN', 'LIST', or 'NONE'</td>
<td>Specifies carriage control processing.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>CONVERT=$form</td>
<td>'LITTLE_ENDIAN', 'BIG_ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG', or 'NATIVE' (default is 'NATIVE')</td>
<td>Specifies a numeric format for unformatted data.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>DEFAULTFILE=$var</td>
<td>Character expression</td>
<td>Specifies a default file pathname string.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>DELIM=$delimiter</td>
<td>'APOSTROPHE', 'QUOTE' or 'NONE' (default is 'NONE')</td>
<td>Specifies the delimiting character for list-directed or namelist data.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>DIRECT=$dir</td>
<td>'NO' or 'YES'</td>
<td>Returns whether file is connected for direct access.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td>DISPOSE=dis (or DISP=dis)</td>
<td>'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', or 'SUBMIT/DELETE' (default is 'DELETE' for scratch files; 'KEEP' for all other files)</td>
<td>Specifies the status of a file after the unit is closed.</td>
<td>OPEN, CLOSE</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>formatlist</td>
<td>Character variable or expression</td>
<td>Lists edit descriptors. Used in FORMAT statements and format specifiers (the FMT=formatspec option) to describe the format of data.</td>
<td>FORMAT PRINT READ, WRITE</td>
</tr>
<tr>
<td>END=endlabel</td>
<td>Integer between 1 and 99999</td>
<td>When an end of file is encountered, transfers control to the statement whose label is specified.</td>
<td>READ</td>
</tr>
<tr>
<td>EOR=eorlabel</td>
<td>Integer between 1 and 99999</td>
<td>When an end of record is encountered, transfers to the statement whose label is specified.</td>
<td>READ</td>
</tr>
<tr>
<td>ERR=errlabel</td>
<td>Integer between 1 and 99999</td>
<td>Specifies the label of an executable statement where execution is transferred after an I/O error.</td>
<td>All except PRINT</td>
</tr>
<tr>
<td><strong>EXIST=ex</strong></td>
<td><strong>.TRUE. or .FALSE.</strong></td>
<td>Returns whether a file exists and can be opened.</td>
<td><strong>INQUIRE</strong></td>
</tr>
<tr>
<td><strong>FILE=file (or NAME=name)</strong></td>
<td>Character variable or expression. Length and format of the name are determined by the operating system</td>
<td>Specifies the name of a file</td>
<td><strong>INQUIRE OPEN</strong></td>
</tr>
<tr>
<td><strong>[FMT=}formatspec</strong></td>
<td>Character variable or expression</td>
<td>Specifies an editlist to use to format data.</td>
<td><strong>PRINT READ, WRITE</strong></td>
</tr>
<tr>
<td><strong>FORM=form</strong></td>
<td>'FORMATTED', 'UNFORMATTED', or 'BINARY'</td>
<td>Specifies a file's format.</td>
<td><strong>INQUIRE OPEN</strong></td>
</tr>
<tr>
<td><strong>FORMATTED=fmt</strong></td>
<td>'NO' or 'YES'</td>
<td>Returns whether a file is connected for formatted data transfer.</td>
<td><strong>INQUIRE</strong></td>
</tr>
<tr>
<td><strong>IOFOCUS=iof</strong></td>
<td>.TRUE. or .FALSE. (default is .TRUE. unless unit '*' is specified)</td>
<td>Specifies whether a unit is the active window in a QuickWin application.</td>
<td><strong>INQUIRE OPEN</strong></td>
</tr>
<tr>
<td><strong>iolist</strong></td>
<td>List of variables of any type, character expression, or NAMELIST</td>
<td>Specifies items to be input or output.</td>
<td><strong>PRINT READ, WRITE</strong></td>
</tr>
<tr>
<td><strong>IOSTAT=iostat</strong></td>
<td>Integer variable</td>
<td>Specifies a variable whose value indicates whether an I/O error has occurred.</td>
<td>All except <strong>PRINT</strong></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Values</td>
<td>Notes</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>MAXREC=var</strong></td>
<td>Numeric expression</td>
<td>Specifies the maximum number of records that can be transferred to or from a direct access file.</td>
<td>OPEN</td>
</tr>
<tr>
<td><strong>MODE=permission</strong></td>
<td>'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')</td>
<td>Same as ACTION.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td><strong>NAMED=var</strong></td>
<td>.TRUE. or .FALSE.</td>
<td>Returns whether a file is named.</td>
<td>INQUIRE</td>
</tr>
<tr>
<td><strong>NEXTREC=nr</strong></td>
<td>Integer variable</td>
<td>Returns where the next record can be read or written in a file.</td>
<td>INQUIRE</td>
</tr>
<tr>
<td><strong>[NML=]nmlspec</strong></td>
<td>Namelist name</td>
<td>Specifies a namelist group to be input or output.</td>
<td>PRINT READ, WRITE</td>
</tr>
<tr>
<td><strong>NUMBER=num</strong></td>
<td>Integer variable</td>
<td>Returns the number of the unit connected to a file.</td>
<td>INQUIRE</td>
</tr>
<tr>
<td><strong>OPENED=od</strong></td>
<td>.TRUE. or .FALSE.</td>
<td>Returns whether a file is connected.</td>
<td>INQUIRE</td>
</tr>
<tr>
<td><strong>ORGANIZATION=org</strong></td>
<td>'SEQUENTIAL' or 'RELATIVE' (default is 'SEQUENTIAL')</td>
<td>Specifies the internal organization of a file.</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td><strong>PAD=pad_switch</strong></td>
<td>'YES' or 'NO' (default is 'YES')</td>
<td>Specifies whether an input record is padded with blanks when the input list or format requires more data than</td>
<td>INQUIRE OPEN</td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>Specification</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>POSITION=\textit{file_pos}</td>
<td>Specifies position in a file.</td>
<td>\text{OPEN}</td>
<td></td>
</tr>
<tr>
<td>READ=\textit{rd}</td>
<td>Returns whether a file can be read.</td>
<td>\text{INQUIRE}</td>
<td></td>
</tr>
<tr>
<td>READWRITE=\textit{rdwr}</td>
<td>Returns whether a file can be both read and written to.</td>
<td>\text{INQUIRE}</td>
<td></td>
</tr>
<tr>
<td>REC=\textit{rec}</td>
<td>Specifies the first (or only) record of a file to be read from, or written to.</td>
<td>\text{READ, WRITE}</td>
<td></td>
</tr>
<tr>
<td>RECL=\textit{length} (or RECORDSIZE=\textit{length})</td>
<td>Specifies the record length in direct access files, or the maximum record length in sequential files.</td>
<td>\text{INQUIRE}</td>
<td></td>
</tr>
<tr>
<td>RECORDTYPE=\textit{typ}</td>
<td>Specifies the type of records in a file.</td>
<td>\text{INQUIRE}</td>
<td></td>
</tr>
<tr>
<td><strong>SEQUENTIAL=seq</strong></td>
<td>'NO' or 'YES'</td>
<td>Returns whether file is connected for sequential access.</td>
<td><strong>INQUIRE</strong></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>SHARE=share</strong></td>
<td>'COMPAT', 'DENYNONE', 'DENYWR', 'DENYRD', or 'DENYRW' (default is 'DENYNONE')</td>
<td>Controls how other processes can simultaneously access a file on networked systems.</td>
<td><strong>INQUIRE OPEN</strong></td>
</tr>
<tr>
<td><strong>SHARED</strong></td>
<td></td>
<td>Specifies that a file is connected for shared access by more than one program executing simultaneously.</td>
<td><strong>OPEN</strong></td>
</tr>
<tr>
<td><strong>SIZE=size</strong></td>
<td>Integer variable</td>
<td>Returns the number of characters read in a nonadvancing READ before an end-of-record condition occurred.</td>
<td><strong>READ</strong></td>
</tr>
<tr>
<td><strong>STATUS=status</strong></td>
<td>'OLD', 'NEW', 'UNKNOWN' or 'SCRATCH' (default is 'UNKNOWN')</td>
<td>Specifies the status of a file on opening and/or closing.</td>
<td><strong>CLOSE OPEN</strong></td>
</tr>
<tr>
<td><strong>TITLE=name</strong></td>
<td>Character expression</td>
<td>Specifies the name of a child window in a QuickWin application.</td>
<td><strong>OPEN</strong></td>
</tr>
<tr>
<td>UNFORMATTED=unf</td>
<td>'NO' or 'YES'</td>
<td>Returns whether a file is connected for unformatted data transfer.</td>
<td>INQUIRE</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>-----------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>[UNIT=]unitspec</td>
<td>Integer variable or expression</td>
<td>Specifies the unit to which a file is connected.</td>
<td>All except PRINT</td>
</tr>
<tr>
<td>USEROPEN=fname</td>
<td>Name of a user-written function</td>
<td>Specifies an external function that controls the opening of a file.</td>
<td>OPEN</td>
</tr>
<tr>
<td>WRITE=rd</td>
<td>'NO' or 'YES'</td>
<td>Returns whether a file can be written to.</td>
<td>INQUIRE</td>
</tr>
</tbody>
</table>

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
- BUFFERED
- 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:**
ACCESS Specifier

The ACCESS specifier asks how a file is connected. It takes the following form:

\[ \text{ACCESS} = acc \]

\( acc \)
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:

\[ \text{ACTION} = act \]

\( act \)
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, W9*)
The **BINARY** specifier asks whether a file is connected to a binary file. It takes the following form:

\[
\text{BINARY} = \text{bin}
\]

*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 non-binary 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:

\[
\text{BLANK} = \text{blnk}
\]

*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**

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

\[
\text{BLOCKSIZE} = \text{bks}
\]

*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 run-time buffering is in effect. It takes the following form:

\[ \text{BUFFERED} = bf \]

- \( bf \) 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:

\[ \text{CARRIAGECONTROL} = cc \]

- \( cc \) 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 Compaq (formerly 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 Compaq 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 Compaq VAX F_floating, D_floating, and H_floating in effect
- **'VAXG'** If the file is connected with little endian integer and Compaq 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:

\[
\text{DELIM} = \text{del}
\]

\[
\text{del}
\]
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:

\[
\text{DIRECT} = \text{dir}
\]

\[
\text{dir}
\]
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:

\[
\text{EXIST} = \text{ex}
\]

\[
\text{ex}
\]
Is a scalar default logical variable that is assigned one of the following values:
values:

.THUE. 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, W9*), formatted, or unformatted data transfer. It takes the following form:

\[
\text{FORM} = fm
\]

\( fm \)

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:

\[
\text{FORMATTED} = fmt
\]

\( fmt \)

Is a scalar default character variable that is assigned one of the following values:
IOFOCUS Specifier (WNT, W9*)

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

\[
\text{IOFOCUS} = \text{iof}
\]

\text{iof}

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

- \text{.TRUE.} If the specified unit is the active window in a QuickWin application
- \text{.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, W9*)

\text{MODE} is a nonstandard synonym for \text{ACTION}.

NAME Specifier

The NAME specifier returns the name of a file. It takes the following form:

\[
\text{NAME} = \text{nme}
\]

\text{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, \text{nme} is undefined.

The value assigned to \text{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:

\[
\text{NAMED} = nmd
\]

\[
nmd
\]

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:

\[
\text{NEXTREC} = nr
\]

\[
 nr
\]

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:
PAD Specifier

The **PAD** specifier asks whether blank padding was specified for the file. It takes the following form:

\[
\text{PAD} = pd
\]

*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:

\[
\text{POSITION} = pos
\]

*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:

\[
\text{READ} = rd
\]

*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:

\[
\text{READWRITE} = rdwr
\]

*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, W9*)

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:
WRITE Specifier

The WRITE specifier asks whether a file can be written to. It takes the following form:

\[
\text{WRITE} = \text{wr}
\]

\text{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 (and contains links to their descriptions):

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Values</th>
<th>Function</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>'SEQUENTIAL' 'DIRECT' 'APPEND'</td>
<td>Access mode</td>
<td>'SEQUENTIAL'</td>
</tr>
<tr>
<td><strong>ACTION</strong> (or MODE ¹)</td>
<td>'READ'</td>
<td>'WRITE'</td>
<td>'READWRITE'</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>ASSOCIATE_VARIABLE</strong></td>
<td>var</td>
<td>Next direct access record</td>
<td>No default</td>
</tr>
<tr>
<td><strong>BLANK</strong></td>
<td>'NULL'</td>
<td>Interpretation of blanks</td>
<td>'NULL'</td>
</tr>
<tr>
<td><strong>BLOCKSIZE</strong></td>
<td>n_expr</td>
<td>Physical block size</td>
<td>Filesystem default</td>
</tr>
<tr>
<td><strong>BUFFERCOUNT</strong></td>
<td>n_expr</td>
<td>Number of I/O buffers</td>
<td>One</td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>'YES'</td>
<td>Buffering for WRITE operations</td>
<td>'NO'</td>
</tr>
<tr>
<td><strong>CARRIAGE_CONTROL</strong></td>
<td>'FORTRAN'</td>
<td>Formatted: 'LIST'² Unformatted: 'NONE'</td>
<td></td>
</tr>
<tr>
<td><strong>CONVERT</strong></td>
<td>'LITTLE_ENDIAN'</td>
<td>Numeric format specification</td>
<td>'NATIVE'</td>
</tr>
<tr>
<td><strong>DEFAULTFILE</strong></td>
<td>c_expr</td>
<td>Default file pathname</td>
<td>Current working directory</td>
</tr>
<tr>
<td><strong>DELIM</strong></td>
<td>'APOSTROPHE'</td>
<td>Delimiter for character constants</td>
<td>'NONE'</td>
</tr>
<tr>
<td><strong>DISPOSE</strong> (or DISP)</td>
<td>'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'</td>
<td>File disposition at close</td>
<td>'KEEP'</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------</td>
<td>--------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>ERR</strong></td>
<td>label</td>
<td>Error transfer control</td>
<td>No default</td>
</tr>
<tr>
<td><strong>FILE</strong> (or NAME)</td>
<td>c_expr</td>
<td>File pathname (file name)</td>
<td>fort.n 3</td>
</tr>
<tr>
<td><strong>FORM</strong></td>
<td>'FORMATTED' 'UNFORMATTED' 'BINARY' 1</td>
<td>Format type</td>
<td>Depends on <strong>ACCESS</strong> setting</td>
</tr>
<tr>
<td><strong>IOFOCUS</strong> 1</td>
<td>.TRUE. or .FALSE.</td>
<td>Active window in QuickWin application</td>
<td>.TRUE. 4</td>
</tr>
<tr>
<td><strong>IOSTAT</strong></td>
<td>var</td>
<td>I/O status</td>
<td>No default</td>
</tr>
<tr>
<td><strong>MAXREC</strong></td>
<td>n_expr</td>
<td>Direct access record limit</td>
<td>No limit</td>
</tr>
<tr>
<td><strong>ORGANIZATION</strong></td>
<td>'SEQUENTIAL' 'RELATIVE'</td>
<td>File organization</td>
<td>'SEQUENTIAL'</td>
</tr>
<tr>
<td><strong>PAD</strong></td>
<td>'YES' 'NO'</td>
<td>Record padding</td>
<td>'YES'</td>
</tr>
<tr>
<td><strong>POSITION</strong></td>
<td>'ASIS' 'REWIND' 'APPEND'</td>
<td>File positioning</td>
<td>'ASIS'</td>
</tr>
<tr>
<td><strong>READONLY</strong></td>
<td>No value</td>
<td>Write protection</td>
<td>No default</td>
</tr>
<tr>
<td><strong>RECL</strong> (or RECORDSIZE)</td>
<td>n_expr</td>
<td>Record length</td>
<td>Depends on <strong>RECORDTYPE, ORGANIZATION</strong>, an <strong>FORM</strong> settings 5</td>
</tr>
</tbody>
</table>
| **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 | 'DENYWR' |
| **SHARE** 6 | No value | File sharing allowed | U*X: SHARED WNT and W9*: No default |
| **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 Tru64 UNIX and Linux 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:

\[
\text{ACCESS} = acc
\]

\( acc \)

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:

\[
\text{ACTION} = act
\]

\( act \)

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
```
blk
Is a scalar default character expression that evaluates to one of the following values:

'NULL' Indicates all blanks are ignored, except for an all-blank field (which has a value of zero).

'ZERO' Indicates all blanks (other than leading blanks) are treated as zeros.

The default is 'NULL' (for explicitly OPENed files, preconnected files, and internal files). If you specify compiler option /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, W9*) 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'LITTLE_ENDIAN' 1</td>
<td>Little endian integer data 2 and IEEE® floating-point data. 3</td>
</tr>
<tr>
<td>'BIG_ENDIAN' 1</td>
<td>Big endian integer data 2 and IEEE floating-point data. 3</td>
</tr>
<tr>
<td>'CRAY'</td>
<td>Big endian integer data 2 and CRAY® floating-point data of size REAL(8) or COMPLEX(8).</td>
</tr>
<tr>
<td>'FDX'</td>
<td>Little endian integer data 2 and Compaq (formerly 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) or COMPLEX(16). 4</td>
</tr>
<tr>
<td>'FGX'</td>
<td>Little endian integer data 2 and Compaq VAX floating-point data of format F_floating for REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16). 4</td>
</tr>
<tr>
<td>'IBM'</td>
<td>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).</td>
</tr>
<tr>
<td>'VAXD'</td>
<td>Little endian integer data 2 and Compaq VAX floating-point data of format F_floating for size REAL (4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16). 4</td>
</tr>
<tr>
<td>'VAXG'</td>
<td>Little endian integer data 2 and Compaq VAX floating-point data of format F_floating for size REAL (4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16). 4</td>
</tr>
<tr>
<td>'NATIVE'</td>
<td>No data conversion. This is the default.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Method Used</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>An environment variable</td>
<td>Highest</td>
</tr>
<tr>
<td><strong>OPEN</strong>(CONVERT=<code>convert</code>)</td>
<td>·</td>
</tr>
<tr>
<td><code>OPTIONS/CONVERT</code></td>
<td>·</td>
</tr>
<tr>
<td>The <code>/convert:keyword</code> compiler option</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

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', 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.
o 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:

```fortran
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:

```fortran
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:

\[
\text{DISPOSE} = dis \\
\text{DISP} = dis
\]

\(dis\)

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:

\[
\text{FILE} = name
\]

\(name\)

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):

```plaintext
myprog first.fil " " third.txt
```

MYPROG contains four **OPEN** statements with blank filenames, in the following order:

```plaintext
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, W9*), 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, W9*)**

The **IOFOCUS** specifier indicates whether a particular unit is the active window in a QuickWin application. It takes the following form:

\[
\text{IOFOCUS} = \text{iof}
\]

**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:

\[ \text{MAXREC} = mr \]

\(mr\)

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, W9*)**

**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:

\[ \text{ORGANIZATION} = \text{org} \]

\(\text{org}\)

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:

\[
\text{PAD} = pd
\]

\(pd\) 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:

```fortran
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, Compaq 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:

\[
\text{POSITION} = pos
\]

\(pos\) 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 compiler option `/assume:byterecl` 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 Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux Systems

<table>
<thead>
<tr>
<th>Record Type</th>
<th>CARRIAGECONTROL</th>
<th>Formatted (size in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length</td>
<td>'NONE'</td>
<td>2147483647 (2**31-1)</td>
</tr>
<tr>
<td>Variable-length</td>
<td>'NONE'</td>
<td>2147483640 (2**31-8)</td>
</tr>
<tr>
<td>Segmented</td>
<td>'NONE'</td>
<td>32764 (2**15-4)</td>
</tr>
<tr>
<td>Stream</td>
<td>'NONE'</td>
<td>2147483647 (2**31-1)</td>
</tr>
<tr>
<td>Stream_CR</td>
<td>'LIST'</td>
<td>2147483647 (2**31-1)</td>
</tr>
<tr>
<td>Stream_LF</td>
<td>'LIST'</td>
<td>2147483647 (2**31-1)</td>
</tr>
</tbody>
</table>

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 Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux Systems

<table>
<thead>
<tr>
<th>RECORDTYPE</th>
<th>RECL value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'FIXED'</td>
<td>None; value must be explicitly specified.</td>
</tr>
<tr>
<td>All other settings</td>
<td>132 bytes for formatted records; 510 longwords for unformatted records.</td>
</tr>
</tbody>
</table>

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
SHARE Specifier (WNT, W9*)

The SHARE specifier indicates whether file locking is implemented while the unit is open. It takes the following form:

\[
\text{SHARE} = \text{shr}
\]

\text{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 'DENYWR'. However, if you specify compiler option \text{/fpscomp:general} or the SHARED specifier, 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
```

On Tru64 UNIX and Linux systems, shared access is the default for the Fortran I/O system. On Windows NT (including Windows 2000) and Windows 9* systems, it is the default if SHARED or compiler option `/fpscomp:general` 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 compiler option /f66 (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 (WNT, W95)

The TITLE specifier indicates the name of a child window in a QuickWin application. It takes the following form:

\[
\text{TITLE} = \text{name}
\]

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 :: FILENAME
!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
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

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.
Compiler Directives

Compaq Fortran provides the following compiler directives:

- **General directives**
  
  Perform general-purpose tasks during compilation.

- **Parallel directives** (TU*X only)
  
  Specify parallel regions, and characteristics of data and threads for shared access of data in memory.

General Compiler Directives

Compaq 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.

- **IVDEP**
  Assists the compiler's dependence analysis.

- **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**
  Affects data alignment and warnings about data alignment.

- **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 language standard
specified on the command line (Fortran 95 or Fortran 90).

- **TITLE** and **SUBTITLE**
  
  Specifies a title or subtitle for a listing header.

- **UNROLL**

  Tells the compiler's optimizer how many times to unroll a DO loop.

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:

```
cDEC$
```

- **c**
  Is one of the following: C (or c), !, or *.

The following are source form rules for directive prefixes:

- Prefixes beginning with C (or c) and * are only allowed in fixed 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.

Parallel Directives for Tru64 UNIX Systems

Compaq Fortran provides two sets of directives that support shared memory parallel programming. These directives provide ways to specify the actions taken by the compiler and run-time system when executing a Fortran program in parallel.

This section describes syntax rules and the following sets of directives:

- **OpenMP Fortran compiler directives**
  These directives comply with OpenMP Fortran Application Program Interface (API) specification Version 1.0.

- **Compaq Fortran parallel compiler directives**
  These directives are provided for compatibility with older programs that were written for parallel execution.

Each set has a specific prefix that identifies the directives in the set. You can only use one set of directives within the same program.

For more information on how to use these directives, see your user manual.

Syntax Rules for Parallel Directives (TU*X only)

The following general syntax rules apply to all parallel compiler directives. You must follow these rules precisely to compile your program properly and obtain
meaningful results.

A parallel directive prefix (tag) takes one of the following forms:

\begin{align*}
\text{c$OMP} \\
\text{c$PAR}
\end{align*}

\begin{align*}
\text{c} \\
\text{Is one of the following: C (or c), !, or *}
\end{align*}

The following are source form rules for directive prefixes:

\begin{itemize}
\item Prefixes beginning with C (or c) and * are only allowed in fixed and tab source forms.

In these source forms, the prefix must appear in columns 1 through 5; column 6 must be a blank or tab. From column 7 on, blanks are insignificant, so the directive can be positioned anywhere on the line after column 6. A directive ends in column 72 (or column 132, if a compiler option is specified).

\item 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.
\end{itemize}

Parallel directives can be continued and comments can be mixed within a continued parallel directive. The directive prefix must appear on each line of a continued directive.

A comment can follow a directive on the same line.

Additional Fortran statements (or directives) cannot appear on the same line as the parallel directive.

Parallel directives cannot appear within a continued Fortran statement.

\section*{Examples}

The following examples are equivalent:

\begin{verbatim}
!$OMP PARALLEL DO &
!$OMP SHARED(A,B,C)
\end{verbatim}
Data Scope Attribute Clauses (TU*X only)

Some of the OpenMP Fortran and Compaq Fortran parallel directives have clauses (or keywords) you can specify to control the scope attributes of variables for the duration of the directive. This section discusses the following clauses:

- COPYIN
- DEFAULT
- FIRSTPRIVATE
- LASTPRIVATE
- PRIVATE
- REDUCTION
- SHARED

Other clauses (or keywords) are available for some parallel directives. For more information, see each directive description.

OpenMP Fortran API Compiler Directives (TU*X only)

Compaq Fortran supplies parallel directives that comply with OpenMP Fortran Application Program Interface (API) specification Version 1.0.

Use these directives when writing new programs for parallel execution.

This section describes conditional compilation rules, nesting and binding rules, and the following directives:

- ATOMIC
  Specifies that a specific memory location is to be updated dynamically.

- BARRIER
  Synchronizes all the threads in a team.

- CRITICAL
  Restricts access for a block of code to only one thread at a time.

- DO
Specifies that the iterations of the immediately following DO loop must be executed in parallel.

- **FLUSH**
  Specifies synchronization points where the implementation must have a consistent view of memory.

- **MASTER**
  Specifies a block of code to be executed by the master thread of the team.

- **ORDERED**
  Specifies a block of code to be executed sequentially.

- **PARALLEL**
  Defines a parallel region.

- **PARALLEL DO**
  Defines a parallel region that contains a single DO directive.

- **PARALLEL SECTIONS**
  Defines a parallel region that contains SECTIONS directives.

- **SECTIONS**
  Specifies a block of code to be divided among threads in a team (a worksharing area).

- **SINGLE**
  Specifies a block of code to be executed by only one thread in a team.

- **THREADPRIVATE**
  Makes named common blocks private to a thread but global within the thread.

The OpenMP parallel directives can be grouped into the categories shown in the following table:

**Categories of OpenMP Fortran Parallel Directives (TU*X only)**
Conditional Compilation Rules (TU*X)

The OpenMP Fortran API lets you conditionally compile Compaq Fortran statements if you use the appropriate directive prefix.

The prefix depends on which source form you are using, although !$ is valid in all forms.

The prefix must be followed by a valid Compaq Fortran statement on the same line.

Fixed Source Form

For fixed source form programs, the conditional compilation prefix is one of the following: !$, C$ (or c$), or *$.

The prefix must start in column one and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.
Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six. For example, the following forms for specifying conditional compilation are equivalent:

```
c23456789
!$ IAM = OMP_GET_THREAD_NUM( ) +
!$ * INDEX

#ifdef _OPENMP
    IAM = OMP_GET_THREAD_NUM( ) +
    * INDEX
#endif
```

**Free Source Form**

The free source form conditional compilation prefix is `!$`. This prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Free-form source rules apply to the directive line.

Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix with optional white space before and after the ampersand.

**For More Information:**

- On how the conditional prefix is interpreted, see your user manual.
- On a macro that can be used to denote conditional compilation, see your user manual.

**Nesting and Binding Rules (TU*X only)**

This section describes the dynamic nesting and binding rules for OpenMP Fortran API directives.

**Binding Rules**

The following rules apply to dynamic binding:

- The **DO**, **SECTIONS**, **SINGLE**, **MASTER**, and **BARRIER** directives bind to the dynamically enclosing **PARALLEL** directive, if one exists.

- The **ORDERED** directive binds to the dynamically enclosing **DO** directive.

- The **ATOMIC** directive enforces exclusive access with respect to **ATOMIC** directives in all threads, not just the current team.
The **CRITICAL** directive enforces exclusive access with respect to **CRITICAL** directives in all threads, not just the current team.

A directive can never bind to any directive outside the closest enclosing **PARALLEL** directive.

**Nesting Rules**

The following rules apply to dynamic nesting:

- A **PARALLEL** directive dynamically inside another **PARALLEL** directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.

- **DO**, **SECTIONS**, and **SINGLE** directives that bind to the same **PARALLEL** directive are not allowed to be nested one inside the other.

- **DO**, **SECTIONS**, and **SINGLE** directives are not permitted in the dynamic extent of **CRITICAL** and **MASTER** directives.

- **BARRIER** directives are not permitted in the dynamic extent of **DO**, **SECTIONS**, **SINGLE**, **MASTER**, and **CRITICAL** directives.

- **MASTER** directives are not permitted in the dynamic extent of **DO**, **SECTIONS**, and **SINGLE** directives.

- **ORDERED** sections are not allowed in the dynamic extent of **CRITICAL** sections.

- Any directive set that is legal when executed dynamically inside a **PARALLEL** region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

**Examples**

The following example shows nested **PARALLEL** regions:

```c
    c$OMP PARALLEL DEFAULT(SHARED)
    c$OMP DO
        DO I =1, N
        c$OMP PARALLEL SHARED(I,N)
        c$OMP DO
            DO J =1, N
                CALL WORK(I,J)
            END DO
        END DO
    c$OMP END PARALLEL
    END DO
```
Note that the inner and outer DO directives bind to different PARALLEL regions.

The following example shows a variation of the preceding example:

```fortran
      c$OMP PARALLEL DEFAULT(SHARED)
      c$OMP DO
          DO I = 1, N
              CALL SOME_WORK(I,N)
          END DO
      c$OMP END PARALLEL
...  
      SUBROUTINE SOME_WORK(I,N)
      c$OMP PARALLEL DEFAULT(SHARED)
      c$OMP DO
          DO J = 1, N
              CALL WORK(I,J)
          END DO
      c$OMP END PARALLEL
      RETURN
      END
```

**Compaq Fortran Parallel Compiler Directives (TU*X only)**

Compaq Fortran supplies parallel directives that you can use for compatibility with older programs that were written for parallel execution.

This section describes the following parallel directives:

- **BARRIER**
  
  Synchronizes all the threads in a team. This directive is the same as the OpenMP Fortran API **BARRIER** directive.

- **CHUNK**

  Sets a default chunksize to adjust the number of iterations assigned to a thread.

- **COPYIN**

  Copies the values of listed data objects from the master thread to **PRIVATE** data objects of the same name in slave threads.

- **CRITICAL SECTION**

  Restricts access to a block of code to only one thread at a time.
- **INSTANCE**
  Specifies the availability of named common blocks.

- **MP_SCHEDTYPE**
  Sets a default run-time scheduling type.

- **PARALLEL**
  Defines a parallel region. This directive is the same as the OpenMP Fortran API `PARALLEL` directive with the following exceptions:
  - No REDUCTION clause is permitted.
  - LOCAL is permitted as an alternative spelling for the PRIVATE clause and SHARE is permitted as an alternative spelling for the SHARED clause.

- **PARALLEL DO**
  Defines a parallel region that contains a single `DO` directive. This directive is the same as the OpenMP Fortran API `PARALLEL DO` directive except that the keyword `DOACROSS` is permitted as an alternative spelling for `PARALLEL DO`.

- **PARALLEL SECTIONS**
  Defines a parallel region that contains `SECTIONS` directives. This directive is the same as the OpenMP Fortran API `PARALLEL SECTIONS` directive.

- **PDO**
  Specifies that the iterations of the immediately following `DO` loop must be executed in parallel.

- **PDONE**
  Terminates the current parallel `DO` loop.

- **PSECTIONS**
  Specifies a block of code to be divided among threads in a team (a worksharing area). This directive is the same as the OpenMP Fortran API `SECTIONS` directive with the following exceptions:
- No REDUCTION clause or LASTPRIVATE clause is permitted.
- LOCAL is permitted as an alternative spelling for the PRIVATE clause.

  o **SINGLE PROCESS**

  Specifies a block of code to be executed by only one thread in a team. This directive is the same as the OpenMP Fortran API `SINGLE` directive except that LOCAL is permitted as an alternative spelling for the PRIVATE clause.

  o **TASKCOMMON**

  Makes named common blocks private to a thread but global within the thread. This directive is the same as the OpenMP Fortran API `THREADPRIVATE` directive except that slashes (/ /) do not have to be used to delimit named common blocks.

The Compaq Fortran parallel directives can be grouped into the categories shown in the following **table**.

**Categories of Compaq Fortran Parallel Directives (TU*X only)**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel region</td>
<td>Defines a parallel region: <strong>PARALLEL</strong></td>
</tr>
<tr>
<td>Work-sharing</td>
<td>Divide the execution of the enclosed block of code among the members of the team that encounter it: <strong>PDO, PSECTIONS, and SINGLE PROCESS</strong></td>
</tr>
<tr>
<td>Combined parallel work-sharing</td>
<td>Shortcut for denoting a parallel region that contains only one work-sharing construct: <strong>PARALLEL DO</strong> and <strong>PARALLEL SECTIONS</strong></td>
</tr>
<tr>
<td>Synchronization</td>
<td>Provide various aspects of synchronization; for example, access to a block of code, or execution order of statements within a block of code: <strong>BARRIER</strong> and <strong>CRITICAL SECTION</strong></td>
</tr>
<tr>
<td>Data Environment</td>
<td>Control the data environment during the execution of parallel constructs: <strong>COPYIN, INSTANCE, and TASKCOMMON</strong></td>
</tr>
<tr>
<td>Execution Defaults</td>
<td>Specify a default chunk or schedule type: <strong>CHUNK</strong> and <strong>MP_SCHEDTYPE</strong></td>
</tr>
</tbody>
</table>
For details on how to use these directives, see your user manual.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only)

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.

<table>
<thead>
<tr>
<th>Compiler directive</th>
<th>Equivalent command-line compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE</td>
<td>/warn:declarations or /4Yd</td>
</tr>
<tr>
<td>NODECLAIMER</td>
<td>/warn:nodeclarations or /4Nd</td>
</tr>
<tr>
<td>DEFINE symbol</td>
<td>/define:symbol or /Dsymbol</td>
</tr>
<tr>
<td>FIXEDFORM LINESIZE option</td>
<td>/extend_source[:option] or /4Loption</td>
</tr>
<tr>
<td>FREEFORM</td>
<td>/free or /nofixed, or /4Yf</td>
</tr>
<tr>
<td>NOFREEFORM</td>
<td>/nofree, /fixed, or /4F</td>
</tr>
<tr>
<td>INTEGER:option</td>
<td>/integer_size:option or /4Ioption</td>
</tr>
<tr>
<td>OBJCOMMENT</td>
<td>/libdir</td>
</tr>
<tr>
<td>PACK:option</td>
<td>/alignment[:option] or /Zpoption</td>
</tr>
<tr>
<td>REAL:option</td>
<td>/real_size:option or /4Roption</td>
</tr>
<tr>
<td>STRICT</td>
<td>/warn:stderrors with /stand:f90 or /4Ys</td>
</tr>
<tr>
<td>NOSTRICT</td>
<td>/4Ns</td>
</tr>
</tbody>
</table>

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.

Scope refers to the area in which a name is recognized. A scoping unit is the program or part of a program in which a name is defined or known. It 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
- Unambiguous generic procedure references
- Resolving procedure references
- Association

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 must be unique. It cannot be used to identify any other global entity in the same executable program.

- Scoping unit (Local scope)

  Entities that are declared within a scoping unit. These entities are local to that scoping unit. The names of local entities are divided into classes (see 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**
<table>
<thead>
<tr>
<th>Entity</th>
<th>Scope</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program units</td>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>Common blocks(^1)</td>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>External procedures</td>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>Intrinsic procedures</td>
<td>Global(^2)</td>
<td></td>
</tr>
<tr>
<td>Module procedures</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Internal procedures</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Dummy procedures</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Statement functions</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Derived types</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Components of derived types</td>
<td>Local</td>
<td>Class II</td>
</tr>
<tr>
<td>Named constants</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Named constructs</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Namelist group names</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Generic identifiers</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Argument keywords in procedures</td>
<td>Local</td>
<td>Class III</td>
</tr>
<tr>
<td>Variables that can be referenced throughout a subprogram</td>
<td>Local</td>
<td>Class I</td>
</tr>
<tr>
<td>Variables that are dummy arguments in statement functions</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td><strong>DO</strong> variables in an implied-do list(^3) of a <strong>DATA</strong> or <strong>FORALL</strong> statement, or an array constructor</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>Intrinsic operators</td>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>Defined operators</td>
<td>Local</td>
<td></td>
</tr>
<tr>
<td>Statement labels</td>
<td>Local</td>
<td></td>
</tr>
</tbody>
</table>
Scoping units can contain other scoping units. For example, the following shows six scoping units:

```fortran
MODULE MOD_1
  ! Scoping unit 1
  ...
  CONTAINS
    ! Scoping unit 1
    FUNCTION FIRST
      TYPE NAME
      ...
      END TYPE NAME
    ...
    CONTAINS
      SUBROUTINE SUB_B
        TYPE PROCESS
        ...
        END TYPE PROCESS
      INTERFACE
        SUBROUTINE SUB_A
          ...
          END SUBROUTINE SUB_A
        END INTERFACE
      END SUBROUTINE SUB_B
    END INTERFACE
  END SUBROUTINE SUB_B
END FUNCTION FIRST
END MODULE
```

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
- References to Specific Names
- References to Nonestablished Names
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

Association allows different program units to access the same value through different names. Entities are associated when each is associated with the same storage location.

There are three kinds of association:
- Name association
- 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

Arguments are the values passed to and from functions and subroutines through
calling program argument lists.

Execution of a procedure reference establishes argument association between
an actual argument and its corresponding dummy argument. The name of a
dummy argument can be different from the name of its associated actual
argument (if any).

When the procedure completes execution, the argument association is
terminated.

For More Information:

For 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 \texttt{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.

\textbf{Note:} Implicit declarations can cause problems for host association. It is recommended that you use \texttt{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.

\section*{Examples}

The following example shows host and use association:

\begin{verbatim}
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                          
    ...                                ! B and Q are available through host association.
    Y = B + C + Q + Z                  ! C is a local entity, explicitly declared. Z
    ...                                ! is available through use association.
END SUBROUTINE CONS
END PROGRAM DEMO
\end{verbatim}

The following example shows how a host and an internal procedure can use host-associated entities:

\begin{verbatim}
program INTERNAL
  ! shows use of internal subroutine and CONTAINS statement
  real a,b,c
\end{verbatim}
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 \( \Rightarrow \) 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
- Array Association

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 Compaq 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 Compaq 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**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Storage Requirements (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>2, 4, or 8 $^1$</td>
</tr>
<tr>
<td>LOGICAL(1)</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL(8) $^2$</td>
<td>8</td>
</tr>
<tr>
<td>INTEGER</td>
<td>2, 4, or 8 $^1$</td>
</tr>
<tr>
<td>INTEGER(1)</td>
<td>1</td>
</tr>
<tr>
<td>INTEGER(2)</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER(4)</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER(8) $^2$</td>
<td>8</td>
</tr>
<tr>
<td>REAL</td>
<td>4, 8, or 16 $^3$</td>
</tr>
<tr>
<td>REAL(4)</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
</tbody>
</table>
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 identified 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 /stand compiler option to have these features flagged.

Note: Compaq Fortran fully supports features deleted from Fortran 95.

This section discusses the following topics:

- Deleted Language Features in Fortran 95
- Obsolescent Language Features in Fortran 95
- Obsolescent Language Features in Fortran 90

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 Compaq 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

Compaq 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.

Compaq Fortran flags these features if you specify the /stand compiler option.

Other methods are suggested to achieve the functionality of the following...
obsolescent features:

- Alternate returns

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program use a **CASE construct** to test the value and perform operations.

- 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.

**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.

**Compaq Fortran flags these features if you specify the** `/stand` **compiler option.**

Other methods are suggested to achieve the functionality of the following obsolescent features:

- **Alternate return (labels in an argument list)**

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program test the value and perform operations, using a computed **GO TO statement** 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.
o 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**).

o **H** edit descriptor

To replace this functionality, it is recommended that you use the character constant edit descriptor (see **Character Constant Editing**).

o **PAUSE** statement

To replace this functionality, it is recommended that you use a **READ statement** that awaits input data.

o 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**).

o 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**.
To facilitate compatibility with older versions of Fortran, Compaq 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 Compaq Fortran POINTER statement
- Record structures

These language features are particularly useful in porting older Fortran programs to Fortran 95/90. However, you should avoid using them in new programs on these systems, and in new programs for which portability to other Fortran 95/90 implementations is important.

**FORTRAN-66 Interpretation of the EXTERNAL Statement**

If you specify compiler option /f66, the EXTERNAL statement is interpreted in a way that was specified by the FORTRAN IV (FORTRAN-66) standard. This interpretation became incompatible with FORTRAN 77 and later revisions of the Fortran standard.

The FORTRAN-66 interpretation of the EXTERNAL statement combines the functionality of the INTRINSIC statement 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 95/90 library functions.

The FORTRAN-66 EXTERNAL statement takes the following form:

```
EXTERNAL [*]v [, [*]v] ...
```

* Specifies that a user-supplied function is to be used instead of a Fortran
95/90 library function having the same name.

\( v \)

Is the name of a subprogram or the name of a dummy argument associated with the name of a subprogram.

**Rules and Behavior**

The FORTRAN-66 `EXTERNAL` statement declares that each name in its list is an external function name. Such a name can then be used as an actual argument to a subprogram, which then can use the corresponding dummy argument in a function reference or CALL statement.

However, when used as an argument, a complete function reference represents a value, not a subprogram name; for example, \( \text{SQRT}(B) \) in \( \text{CALL SUBR}(A, \text{SQRT}(B), C) \). 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:

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL SIN, COS, *TAN, SINDEG</td>
<td>SUBROUTINE TRIG(X,F,Y)</td>
</tr>
<tr>
<td>.</td>
<td>Y = F(X)</td>
</tr>
<tr>
<td>.</td>
<td>RETURN</td>
</tr>
<tr>
<td>CALL TRIG(ANGLE, SIN, SINE)</td>
<td>END</td>
</tr>
<tr>
<td>.</td>
<td>FUNCTION TAN(X)</td>
</tr>
<tr>
<td>.</td>
<td>TAN = SIN(X)/COS(X)</td>
</tr>
<tr>
<td>CALL TRIG(ANGLE, COS, COSINE)</td>
<td>RETURN</td>
</tr>
<tr>
<td>.</td>
<td>END</td>
</tr>
<tr>
<td>CALL TRIG(ANGLE, TAN, TANGNT)</td>
<td>FUNCTION SINDEG(X)</td>
</tr>
<tr>
<td>.</td>
<td>SINDEG = SIN(X*3.1459/180)</td>
</tr>
<tr>
<td>.</td>
<td>RETURN</td>
</tr>
<tr>
<td>CALL TRIG(ANGLED, SINDEG, SINE)</td>
<td>END</td>
</tr>
</tbody>
</table>

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:

\[ Y = \text{SIN}(X) \]
\[ Y = \text{COS}(X) \]
\[ Y = \text{TAN}(X) \]
\[ Y = \text{SINDEG}(X) \]

The functions `SIN` and `COS` are examples of trigonometric functions supplied in
the Fortran 95/90 library. The function TAN is also supplied in the library, but the asterisk (*) in the \texttt{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 \texttt{PARAMETER} statement discussed here is similar to the one discussed in \texttt{PARAMETER}; they both assign a name to a constant. However, this \texttt{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 \texttt{PARAMETER} statement takes the following form:

\begin{verbatim}
PARAMETER \texttt{c} = \texttt{expr} \[, \texttt{c} = \texttt{expr}] ...
\end{verbatim}

- \texttt{c}
  - Is the name of the constant.

- \texttt{expr}
  - Is an initialization expression. It can be of any data type.

**Rules and Behavior**

Each name \texttt{c} becomes a constant and is defined as the value of expression \texttt{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:

\begin{verbatim}
PARAMETER I=3
PARAMETER M=I.25 \hspace{1cm} ! Not allowed
PARAMETER N=(1.703, I) \hspace{1cm} ! Allowed
\end{verbatim}

The name used in the \texttt{PARAMETER} statement identifies only the name's corresponding constant in that program unit. Such a name can be defined only once in \texttt{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:

```fortran
PARAMETER PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

**For More Information:**

For details on compile-time constant expressions, see `PARAMETER`.

**Alternative Syntax for Octal and Hexadecimal Constants**

In Compaq Fortran, you can use an alternative syntax for octal and hexadecimal constants. The following table shows this alternative syntax and equivalents:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Alternative Syntax</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octal</td>
<td>'0..7'O</td>
<td>O'0..7'</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>'0..F'X</td>
<td>Z'0..F'</td>
</tr>
</tbody>
</table>

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 Compaq 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 Compaq 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 Compaq 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 Compaq Fortran as a
language extension. It is still supported in Visual Fortran, although its
functionality has been replaced by standard Fortran 95/90 derived types. Record
structures in existing code can be easily converted to Fortran 95/90 derived
type structures for portability, but can also be left in their old form. In most
cases, a Compaq Fortran record and a Fortran 95/90 derived type can be used
interchangeably.

Compaq Fortran record structures are similar to Fortran 95/90 derived types.

A **record structure** is an aggregate entity containing one or more elements.
(Record elements are also called fields or components.) You can use records
when you need to declare and operate on multi-field data structures in your
programs.

Creating a record is a two-step process:

1. You must define the form of the record with a multistatement **structure
declaration**.

2. You must use a **RECORD** statement to declare the record as an entity with
a name. (More than one **RECORD** statement can refer to a given
Examples

Compaq Fortran record structures, using only intrinsic types, easily convert to Fortran 95/90 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:

<table>
<thead>
<tr>
<th>Record Structure</th>
<th>Fortran 95/90 Derived-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRUCTURE</strong> /employee_name/</td>
<td><strong>TYPE</strong> employee_name</td>
</tr>
<tr>
<td>CHARACTER*25 last_name</td>
<td>CHARACTER*25 last_name</td>
</tr>
<tr>
<td>CHARACTER*15 first_name</td>
<td>CHARACTER*15 first_name</td>
</tr>
<tr>
<td><strong>END STRUCTURE</strong></td>
<td><strong>END TYPE</strong></td>
</tr>
<tr>
<td><strong>STRUCTURE</strong> /employee_addr/</td>
<td><strong>TYPE</strong> employee_addr</td>
</tr>
<tr>
<td>CHARACTER*20 street_name</td>
<td>CHARACTER*20 street_name</td>
</tr>
<tr>
<td>INTEGER(2) street_number</td>
<td>INTEGER(2) street_number</td>
</tr>
<tr>
<td>INTEGER(2) apt_number</td>
<td>INTEGER(2) apt_number</td>
</tr>
<tr>
<td>CHARACTER*20 city</td>
<td>CHARACTER*20 city</td>
</tr>
<tr>
<td>CHARACTER*2 state</td>
<td>CHARACTER*2 state</td>
</tr>
<tr>
<td>INTEGER(4) zip</td>
<td>INTEGER(4) zip</td>
</tr>
<tr>
<td><strong>END STRUCTURE</strong></td>
<td><strong>END TYPE</strong></td>
</tr>
</tbody>
</table>

The record structures 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:

<table>
<thead>
<tr>
<th>Record Structure</th>
<th>Fortran 95/90 Derived-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRUCTURE</strong> /employee_data/</td>
<td><strong>TYPE</strong> employee_data</td>
</tr>
<tr>
<td>RECORD /employee_name/ name</td>
<td>TYPE (employee_name) name</td>
</tr>
<tr>
<td>RECORD /employee_addr/ addr</td>
<td>TYPE (employee_addr) addr</td>
</tr>
<tr>
<td>INTEGER(4) telephone</td>
<td>INTEGER(4) telephone</td>
</tr>
<tr>
<td>INTEGER(2) date_of_birth</td>
<td>INTEGER(2) date_of_birth</td>
</tr>
<tr>
<td>INTEGER(2) date_of_hire</td>
<td>INTEGER(2) date_of_hire</td>
</tr>
<tr>
<td>INTEGER(2) social_security(3)</td>
<td>INTEGER(2) social_security(3)</td>
</tr>
<tr>
<td>LOGICAL(2) married</td>
<td>LOGICAL(2) married</td>
</tr>
<tr>
<td>INTEGER(2) dependents</td>
<td>INTEGER(2) dependents</td>
</tr>
<tr>
<td><strong>END STRUCTURE</strong></td>
<td><strong>END TYPE</strong></td>
</tr>
</tbody>
</table>

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.
Additional Language Features

- **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. Compaq Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

Compaq 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, Compaq 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 (including Windows 2000) and Windows 9* systems. Other character sets are available on OpenVMS, Tru64 UNIX, and Linux systems; for details, see the printed Compaq Fortran Language Reference Manual.

For details on the Fortran 95/90 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".

<table>
<thead>
<tr>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>^@ 0</td>
<td>00</td>
<td>NULL</td>
<td>32, 20</td>
<td>ASCII</td>
</tr>
<tr>
<td>^A 1</td>
<td>01</td>
<td>CONTROL</td>
<td>33, 21</td>
<td>41, A</td>
</tr>
<tr>
<td>^B 2</td>
<td>02</td>
<td>QUOTE</td>
<td>34, 22</td>
<td>&quot;</td>
</tr>
<tr>
<td>^C 3</td>
<td>03</td>
<td>POUND</td>
<td>35, 23</td>
<td>#</td>
</tr>
<tr>
<td>^D 4</td>
<td>04</td>
<td>DOLLAR</td>
<td>36, 24</td>
<td>$</td>
</tr>
<tr>
<td>^E 5</td>
<td>05</td>
<td>PERCENT</td>
<td>37, 25</td>
<td>%</td>
</tr>
<tr>
<td>^F 6</td>
<td>06</td>
<td>RSVP</td>
<td>38, 26</td>
<td>&amp;</td>
</tr>
<tr>
<td>^G 7</td>
<td>07</td>
<td>GRACE</td>
<td>39, 27</td>
<td>'</td>
</tr>
<tr>
<td>^H 8</td>
<td>08</td>
<td>BS</td>
<td>40, 28</td>
<td>(</td>
</tr>
<tr>
<td>^I 9</td>
<td>09</td>
<td>EXCLAM</td>
<td>41, 29</td>
<td>)</td>
</tr>
<tr>
<td>^J 10</td>
<td>0A</td>
<td>LF</td>
<td>42, 2A</td>
<td>,</td>
</tr>
<tr>
<td>^K 11</td>
<td>0B</td>
<td>VT</td>
<td>43, 2B</td>
<td>+</td>
</tr>
<tr>
<td>^L 12</td>
<td>0C</td>
<td>FF</td>
<td>44, 2C</td>
<td>,</td>
</tr>
<tr>
<td>^M 13</td>
<td>0D</td>
<td>CR</td>
<td>45, 2D</td>
<td>-</td>
</tr>
<tr>
<td>^N 14</td>
<td>0E</td>
<td>SO</td>
<td>46, 2E</td>
<td>,</td>
</tr>
<tr>
<td>^O 15</td>
<td>0F</td>
<td>SI</td>
<td>47, 2F</td>
<td>/</td>
</tr>
<tr>
<td>^P 16</td>
<td>10</td>
<td>SME</td>
<td>48, 30</td>
<td>0</td>
</tr>
<tr>
<td>^Q 17</td>
<td>11</td>
<td>CS1</td>
<td>49, 31</td>
<td>1</td>
</tr>
<tr>
<td>^R 18</td>
<td>12</td>
<td>DC2</td>
<td>50, 32</td>
<td>2</td>
</tr>
<tr>
<td>^S 19</td>
<td>13</td>
<td>DC3</td>
<td>51, 33</td>
<td>3</td>
</tr>
<tr>
<td>^T 20</td>
<td>14</td>
<td>DC4</td>
<td>52, 34</td>
<td>4</td>
</tr>
<tr>
<td>^U 21</td>
<td>15</td>
<td>NAK</td>
<td>53, 35</td>
<td>5</td>
</tr>
<tr>
<td>^V 22</td>
<td>16</td>
<td>SYM</td>
<td>54, 36</td>
<td>6</td>
</tr>
<tr>
<td>^W 23</td>
<td>17</td>
<td>E1B</td>
<td>55, 37</td>
<td>7</td>
</tr>
<tr>
<td>^X 24</td>
<td>18</td>
<td>CAN</td>
<td>56, 38</td>
<td>8</td>
</tr>
<tr>
<td>^Y 25</td>
<td>19</td>
<td>EM</td>
<td>57, 39</td>
<td>9</td>
</tr>
<tr>
<td>^Z 26</td>
<td>1A</td>
<td>SPACE</td>
<td>58, 3A</td>
<td>;</td>
</tr>
<tr>
<td>^[ 27</td>
<td>1B</td>
<td>ESC</td>
<td>59, 3B</td>
<td></td>
</tr>
<tr>
<td>^\ 28</td>
<td>1C</td>
<td>FS</td>
<td>60, 3C</td>
<td>{</td>
</tr>
<tr>
<td>^] 29</td>
<td>1D</td>
<td>RES</td>
<td>61, 3D</td>
<td>^</td>
</tr>
<tr>
<td>^` 30</td>
<td>1E</td>
<td>BS</td>
<td>62, 3E</td>
<td>\</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
<th>Ctrl Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASCII Character Codes Chart 1
ASCII Character Codes Chart 2 (IBM character set)

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.
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Scan Code</th>
<th>ASCII or Extended</th>
<th>ASCII or Extended with SHIFT</th>
<th>ASCII or Extended with CTRL</th>
<th>ASCII or Extended with ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC</td>
<td>1 01</td>
<td>27 1B ESC</td>
<td>27 1B ESC</td>
<td>27 1B ESC</td>
<td>27 1B ESC</td>
</tr>
<tr>
<td>!</td>
<td>2 02</td>
<td>49 31 1</td>
<td>33 21 !</td>
<td>120 78 NUL</td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>3 03</td>
<td>50 32 2</td>
<td>64 40 !</td>
<td>3 03 NUL</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>4 04</td>
<td>51 33 3</td>
<td>35 23 #</td>
<td>122 7A NUL</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>5 05</td>
<td>52 34 4</td>
<td>36 24 $</td>
<td>123 7B NUL</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>6 06</td>
<td>53 35 5</td>
<td>37 25 %</td>
<td>124 7C NUL</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>7 07</td>
<td>54 36 6</td>
<td>94 SE</td>
<td>30 1E RS</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>8 08</td>
<td>55 37 7</td>
<td>38 26 &amp;</td>
<td>126 7E NUL</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>9 09</td>
<td>56 38 8</td>
<td>42 2A *</td>
<td>127 7F NUL</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>10 0A</td>
<td>57 39 9</td>
<td>40 28 (</td>
<td>128 80 NUL</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>11 0B</td>
<td>48 30 0</td>
<td>41 29 )</td>
<td>129 81 NUL</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>12 0C</td>
<td>45 2D +</td>
<td>95 5F +</td>
<td>31 1F US</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>13 0D</td>
<td>61 2D =</td>
<td>43 2B =</td>
<td>131 83 NUL</td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>14 0E</td>
<td>8 08</td>
<td>8 08</td>
<td>127 7F</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>15 0F</td>
<td>9 09</td>
<td>15 0F NUL</td>
<td>148 94 NUL</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>16 10</td>
<td>117 7I</td>
<td>81 51 Q</td>
<td>17 11 DCI</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>17 11</td>
<td>119 7L</td>
<td>87 57 E</td>
<td>23 17 ED</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>18 12</td>
<td>101 65</td>
<td>69 43 E</td>
<td>5 05 ENQ</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>19 13</td>
<td>114 7T</td>
<td>82 52 R</td>
<td>18 12 DC</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>20 14</td>
<td>114 7S</td>
<td>94 54 T</td>
<td>20 14 ISO</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>21 15</td>
<td>121 7O</td>
<td>89 59 U</td>
<td>25 19 EM</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>22 16</td>
<td>117 7P</td>
<td>85 55 U</td>
<td>21 15 HAK</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>23 17</td>
<td>105 69</td>
<td>73 40 T</td>
<td>5 06 TAB</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>24 18</td>
<td>111 66</td>
<td>79 4F O</td>
<td>15 0E SI</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>25 19</td>
<td>112 7S</td>
<td>80 50 P</td>
<td>16 10 DLE</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>26 1A</td>
<td>91 5B</td>
<td>123 7B {</td>
<td>27 1B ESC</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>27 1B</td>
<td>95 5D</td>
<td>125 7D }</td>
<td>29 1D OS</td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td>28 1C</td>
<td>13 0D CR</td>
<td>13 0D CR</td>
<td>10 0A LF</td>
<td></td>
</tr>
<tr>
<td>LCTRL</td>
<td>29 1D</td>
<td>13 0D CR</td>
<td>13 0D CR</td>
<td>10 0A LF</td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td>30 1E</td>
<td>97 61 a</td>
<td>65 41 A</td>
<td>1 01 SCH</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>31 1F</td>
<td>113 75</td>
<td>83 33 A</td>
<td>19 15 DCX</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>32 20</td>
<td>100 64</td>
<td>68 44 D</td>
<td>4 04 D2X</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>33 21</td>
<td>102 65</td>
<td>70 46 F</td>
<td>6 06 ACK</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>34 22</td>
<td>103 67</td>
<td>71 47 G</td>
<td>7 07 DEL</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>35 23</td>
<td>104 68</td>
<td>72 48 H</td>
<td>8 08 BS</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>36 24</td>
<td>106 69</td>
<td>74 4A J</td>
<td>10 0A LF</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>37 25</td>
<td>107 68</td>
<td>75 4B K</td>
<td>11 0B Y</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>38 26</td>
<td>108 6C</td>
<td>76 4C L</td>
<td>12 0C FF</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>39 27</td>
<td>59 3B</td>
<td>58 3A</td>
<td>39 27 NUL</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>40 28</td>
<td>30 20</td>
<td>54 22</td>
<td>40 28 NUL</td>
<td></td>
</tr>
<tr>
<td>_</td>
<td>41 29</td>
<td>96 40</td>
<td>126 7E _</td>
<td>41 29 NUL</td>
<td></td>
</tr>
<tr>
<td>L SHIFT</td>
<td>42 2A</td>
<td>Z</td>
<td>92 5C \</td>
<td>124 7C</td>
<td>28 1C FS</td>
</tr>
<tr>
<td>`</td>
<td>43 2B</td>
<td>122 7A \</td>
<td>90 5A \</td>
<td>26 1A SUB</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>44 2C</td>
<td>120 78</td>
<td>88 58 V</td>
<td>24 18 CAN</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>45 2D</td>
<td>99 63</td>
<td>67 43 C</td>
<td>3 05 L1X</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>47 2F</td>
<td>118 76</td>
<td>86 56 V</td>
<td>22 16 SYN</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>48 30</td>
<td>96 62</td>
<td>66 42 E</td>
<td>2 02 SX</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>49 31</td>
<td>110 6E</td>
<td>78 4E M</td>
<td>14 0E SO</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>50 32</td>
<td>109 6D</td>
<td>77 4D M</td>
<td>13 0D UK</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>51 33</td>
<td>44 2C</td>
<td>60 3C J</td>
<td>10 0A LF</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>52 34</td>
<td>46 2E</td>
<td>62 3E &gt;</td>
<td>52 34 NUL</td>
<td></td>
</tr>
</tbody>
</table>

### Key Codes Chart 2

<table>
<thead>
<tr>
<th>Key</th>
<th>Scan Code</th>
<th>ASCII or Extended</th>
<th>ASCII or Extended with SHIFT</th>
<th>ASCII or Extended with CTRL</th>
<th>ASCII or Extended with ALL</th>
</tr>
</thead>
</table>
### Character and Key Code Charts

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Dec</th>
<th>Hex</th>
<th>Chars</th>
<th>Dec</th>
<th>Hex</th>
<th>Chars</th>
<th>Dec</th>
<th>Hex</th>
<th>Chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>35</td>
<td>47</td>
<td>2F</td>
<td>✔</td>
<td>63</td>
<td>3F</td>
<td>❌</td>
<td>149</td>
<td>95</td>
<td>NUL</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>47</td>
<td>2F</td>
<td>✔</td>
<td>63</td>
<td>3F</td>
<td>❌</td>
<td>164</td>
<td>A5</td>
<td>NUL</td>
</tr>
<tr>
<td>R</td>
<td>Shift</td>
<td>54</td>
<td>36</td>
<td>✔</td>
<td>55</td>
<td>37</td>
<td>✔</td>
<td>P</td>
<td>Prisc</td>
<td>16</td>
</tr>
<tr>
<td>L</td>
<td>Alt</td>
<td>56</td>
<td>38</td>
<td>✔</td>
<td>57</td>
<td>39</td>
<td>✔</td>
<td></td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>Caps</td>
<td>58</td>
<td>3A</td>
<td>✔</td>
<td>59</td>
<td>3B</td>
<td>NUL</td>
<td>94</td>
<td>54</td>
<td>NUL</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>60</td>
<td>3C</td>
<td>✔</td>
<td>61</td>
<td>3D</td>
<td>✔</td>
<td>62</td>
<td>3E</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>✔</td>
<td>64</td>
<td>40</td>
<td>✔</td>
<td>65</td>
<td>41</td>
<td>✔</td>
<td>66</td>
<td>42</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>✔</td>
<td>68</td>
<td>44</td>
<td>✔</td>
<td>69</td>
<td>45</td>
<td>✔</td>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>NUM</td>
<td>70</td>
<td>46</td>
<td>✔</td>
<td>71</td>
<td>47</td>
<td>✔</td>
<td>72</td>
<td>48</td>
<td>✔</td>
<td>73</td>
</tr>
<tr>
<td>SCROLL</td>
<td>74</td>
<td>4A</td>
<td>✔</td>
<td>75</td>
<td>4B</td>
<td>✔</td>
<td>76</td>
<td>4C</td>
<td>✔</td>
<td>77</td>
</tr>
<tr>
<td>Home</td>
<td>78</td>
<td>4E</td>
<td>✔</td>
<td>79</td>
<td>4F</td>
<td>✔</td>
<td>80</td>
<td>50</td>
<td>✔</td>
<td>81</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>✔</td>
<td>83</td>
<td>53</td>
<td>✔</td>
<td>84</td>
<td>54</td>
<td>✔</td>
<td>85</td>
<td>55</td>
</tr>
</tbody>
</table>

† Extended code return (NUL) or E0 (decimal 20H) as the initial character. This is a signal that a second (extended) code is available in the next stroke buffer.

‡ These keys combinations are only recognized on extended keyboards.

§ These keys are only available on extended keyboards. Most are in the Cursor Control cluster. If the scan code is read from the keyboard port (06H), it appears as two bytes (EDH) followed by the normal scan code. However, when the key that causes the interrupt, only gives one-byte scan code.

|| Added under MS-DOS, SEENT - PRISC 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:

- The model for Integer Data
- The model for Real Data
- The model for Bit Data

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:

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE S_floating</td>
<td>24</td>
</tr>
<tr>
<td>Compaq (formerly DIGITAL) VAX F_floating</td>
<td>24</td>
</tr>
<tr>
<td>IEEE T_floating</td>
<td>53</td>
</tr>
<tr>
<td>Compaq VAX D_floating</td>
<td>53</td>
</tr>
<tr>
<td>Compaq VAX G_floating</td>
<td>53</td>
</tr>
</tbody>
</table>

1VMS only
2The 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 Compaq Fortran.

- \( e \) is an integer in the range \( e_{\text{min}} \) to \( e_{\text{max}} \) inclusive. This range differs depending on the real format, as follows:

<table>
<thead>
<tr>
<th>Model Type</th>
<th>( e_{\text{min}} )</th>
<th>( e_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE S_floating</td>
<td>-125</td>
<td>128</td>
</tr>
<tr>
<td>Compaq VAX F_floating</td>
<td>-127</td>
<td>127</td>
</tr>
<tr>
<td>IEEE T_floating</td>
<td>-1021</td>
<td>1024</td>
</tr>
</tbody>
</table>
\( f_k \) 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:

\[
x = 0 \\
x = s \times 2^e \times \left[ \frac{1}{2} + \sum_{k=1}^{\frac{24}{b}} f_k \times 2^{-k} \right], -125 \leq e \leq 128
\]

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:

\[ j = (w_0 \times 2^0) + (w_1 \times 2^1) + (w_2 \times 2^2) + (w_3 \times 2^3) \]

\[ j = 1 + 0 + 0 + 8 \]

\[ j = 9 \]
Run-Time Library Routines on Tru64 UNIX and Linux Systems

Compaq Fortran provides the following routines for Tru64 UNIX and Linux systems:

- General routines
- Parallel routines

These routines are only available on Tru64 UNIX systems.

General Library Routines (U*X)

Compaq Fortran provides library routines you can use for general purposes on Tru64 UNIX and Linux systems.

The following table summarizes these routines, which are also listed in reference page intro(3f).

For more information on a specific routine, see the appropriate reference page; for example, for more information on alarm, see alarm(3f). If a routine is described on a differently named reference page, it is specified in the tables.

Summary of General Routines (U*X)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Terminates a program (abnormally).</td>
</tr>
<tr>
<td>access</td>
<td>Determines accessibility of a file.</td>
</tr>
<tr>
<td>alarm</td>
<td>Executes a subroutine after a specified time.</td>
</tr>
<tr>
<td>and</td>
<td>Returns bitwise AND of two operands. See bit(3f).</td>
</tr>
<tr>
<td>besj0</td>
<td>Returns a REAL*4 bessel function (kind:1, order:0). See bessel(3f).</td>
</tr>
<tr>
<td>besj1</td>
<td>Returns a REAL*4 bessel function (kind:1, order:1). See bessel(3f).</td>
</tr>
<tr>
<td>besjn</td>
<td>Returns a REAL*4 bessel function (kind:1, order:n). See bessel(3f).</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>bessel</strong></td>
<td>Returns bessel functions.</td>
</tr>
<tr>
<td><strong>besy0</strong></td>
<td>Returns a REAL*4 bessel function (kind:2, order:0). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>besy1</strong></td>
<td>Returns a REAL*4 bessel function (kind:2, order:1). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>besyn</strong></td>
<td>Returns a REAL*4 bessel function (kind:2, order:n). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>bit</strong></td>
<td>Returns bitwise functions.</td>
</tr>
<tr>
<td><strong>chdir</strong></td>
<td>Changes the default directory.</td>
</tr>
<tr>
<td><strong>chmod</strong></td>
<td>Changes the mode of a file.</td>
</tr>
<tr>
<td><strong>ctime</strong></td>
<td>Returns the system time. See <strong>time</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesj0</strong></td>
<td>Returns a REAL*8 bessel function (kind:1, order:0). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesj1</strong></td>
<td>Returns a REAL*8 bessel function (kind:1, order:1). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesjn</strong></td>
<td>Returns a REAL*8 bessel function (kind:1, order:n). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesy0</strong></td>
<td>Returns a REAL*8 bessel function (kind:2, order:0). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesy1</strong></td>
<td>Returns a REAL*8 bessel function (kind:2, order:1). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>dbesyn</strong></td>
<td>Returns a REAL*8 bessel function (kind:2, order:n). See <strong>bessel</strong>(3f).</td>
</tr>
<tr>
<td><strong>derf</strong></td>
<td>Returns a REAL*8 error function. See <strong>erf</strong>(3f).</td>
</tr>
<tr>
<td><strong>derfc</strong></td>
<td>Returns a complimentary REAL*8 error function. See <strong>erf</strong>(3f).</td>
</tr>
<tr>
<td><strong>dffrac</strong></td>
<td>Returns fractional accuracy of a REAL*8 floating-point value. See <strong>flmin</strong>(3f).</td>
</tr>
<tr>
<td><strong>dflmax</strong></td>
<td>Returns the maximum positive REAL*8 floating-point value. See <strong>flmin</strong>(3f).</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>dflmin</td>
<td>Returns the minimum positive REAL*8 floating-point value. See flmin(3f).</td>
</tr>
<tr>
<td>drand</td>
<td>Generates a random number (use drandm instead). See rand(3f).</td>
</tr>
<tr>
<td>drandm</td>
<td>Generates a REAL*8 random number. See random(3f).</td>
</tr>
<tr>
<td>dtime</td>
<td>Returns elapsed delta execution time. See etime(3f).</td>
</tr>
<tr>
<td>erf</td>
<td>Returns a REAL*4 error function.</td>
</tr>
<tr>
<td>erfc</td>
<td>Returns a complimentary REAL*4 error function. See erf(3f).</td>
</tr>
<tr>
<td>etime</td>
<td>Returns elapsed execution time.</td>
</tr>
<tr>
<td>falloc</td>
<td>Allocates memory for an array. See malloc(3f).</td>
</tr>
<tr>
<td>fdate</td>
<td>Returns date and time in an ASCII string.</td>
</tr>
<tr>
<td>ffrac</td>
<td>Returns the fractional accuracy of a REAL*4 floating-point value. See flmin(3f).</td>
</tr>
<tr>
<td>fgetc</td>
<td>Returns a character from a logical unit. See getc(3f).</td>
</tr>
<tr>
<td>flmax</td>
<td>Returns the maximum positive REAL*4 floating-point value. See flmin(3f).</td>
</tr>
<tr>
<td>flmin</td>
<td>Returns the minimum positive REAL*4 floating-point value.</td>
</tr>
<tr>
<td>flush</td>
<td>Writes output to a logical unit.</td>
</tr>
<tr>
<td>for_get_fpe</td>
<td>Returns floating-point exception flags.</td>
</tr>
<tr>
<td>for_rtl_finish_</td>
<td>Initializes the Fortran run-time environment. See for_rtl_init_(3f).</td>
</tr>
<tr>
<td>for_rtl_init_</td>
<td>Cleans up the Fortran run-time environment.</td>
</tr>
<tr>
<td>for_set_fpe</td>
<td>Sets floating-point exception flags. See for_get_fpe(3f).</td>
</tr>
<tr>
<td>for_set_reentrancy</td>
<td>Sets reentrancy protection for the Fortran RTL.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>fork</code></td>
<td>Creates a copy of a calling process.</td>
</tr>
<tr>
<td><code>fputc</code></td>
<td>Writes a character to a logical unit. See <code>putc(3f)</code>.</td>
</tr>
<tr>
<td><code>free</code></td>
<td>Frees memory allocated by malloc or falloc. See <code>malloc(3f)</code>.</td>
</tr>
<tr>
<td><code>fseek</code></td>
<td>Repositions a file on a logical unit.</td>
</tr>
<tr>
<td><code>fstat</code></td>
<td>Returns file status. See <code>stat(3f)</code>.</td>
</tr>
<tr>
<td><code>fsync</code></td>
<td>Writes logical unit changes to permanent storage.</td>
</tr>
<tr>
<td><code>ftell</code></td>
<td>Returns the position of a file associated with a logical unit. See <code>fseek(3f)</code>.</td>
</tr>
<tr>
<td><code>gerror</code></td>
<td>Writes system error messages. See <code>perror(3f)</code>.</td>
</tr>
<tr>
<td><code>getarg</code></td>
<td>Returns command line arguments.</td>
</tr>
<tr>
<td><code>getc</code></td>
<td>Returns a character from a logical unit.</td>
</tr>
<tr>
<td><code>getcwd</code></td>
<td>Returns the pathname of the current working directory.</td>
</tr>
<tr>
<td><code>getenv</code></td>
<td>Returns the value of environment variables.</td>
</tr>
<tr>
<td><code>getfd</code></td>
<td>Returns the file descriptor associated with a Fortran logical unit.</td>
</tr>
<tr>
<td><code>getgid</code></td>
<td>Returns the group ID of the caller. See <code>getuid(3f)</code>.</td>
</tr>
<tr>
<td><code>getlog</code></td>
<td>Returns the user's login name.</td>
</tr>
<tr>
<td><code>getpid</code></td>
<td>Returns the process ID. See <code>getuid(3f)</code>.</td>
</tr>
<tr>
<td><code>getuid</code></td>
<td>Returns the user ID of the caller.</td>
</tr>
<tr>
<td><code>gmtime</code></td>
<td>Returns the GMT system time as month, day, and so forth. See <code>time(3f)</code>.</td>
</tr>
<tr>
<td><code>iargc</code></td>
<td>Returns the index of the last command line argument. See <code>getarg(3f)</code>.</td>
</tr>
<tr>
<td><code>idate</code></td>
<td>Returns the date in numerical form.</td>
</tr>
<tr>
<td><code>ierrno</code></td>
<td>Returns the system error number for the last error. See <code>perror(3f)</code>.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>index</td>
<td>Returns the position of a substring within a string.</td>
</tr>
<tr>
<td>inmax</td>
<td>Returns the maximum positive integer value. See flmin(3f).</td>
</tr>
<tr>
<td>irand</td>
<td>Returns random values (use irandm instead). See rand(3f).</td>
</tr>
<tr>
<td>irandm</td>
<td>Generates an integer random number. See random (3f).</td>
</tr>
<tr>
<td>isatty</td>
<td>Finds the name of a terminal port. See ttynam(3f).</td>
</tr>
<tr>
<td>itime</td>
<td>Returns the time in numerical form. See idate(3f).</td>
</tr>
<tr>
<td>kill</td>
<td>Sends a signal to a process.</td>
</tr>
<tr>
<td>len</td>
<td>Returns the length of a string. See index(3f).</td>
</tr>
<tr>
<td>link</td>
<td>Makes a link to an existing file.</td>
</tr>
<tr>
<td>linblnk</td>
<td>Returns the position of the last non-blank string character. See index(3f).</td>
</tr>
<tr>
<td>loc</td>
<td>Returns the address of an object.</td>
</tr>
<tr>
<td>long</td>
<td>Converts INTEGER<em>2 to INTEGER</em>4.</td>
</tr>
<tr>
<td>lshift</td>
<td>Shifts a word left by ( n ) bits. See bit(3f).</td>
</tr>
<tr>
<td>lstat</td>
<td>Returns information about a file or a link. See stat(3f).</td>
</tr>
<tr>
<td>ltime</td>
<td>Returns the local zone system time as month, day, and so forth. See time(3f).</td>
</tr>
<tr>
<td>malloc</td>
<td>Returns the address of a block of memory.</td>
</tr>
<tr>
<td>not</td>
<td>Returns the bitwise NOT (complement) of an operand. See bit(3f).</td>
</tr>
<tr>
<td>or</td>
<td>Returns the bitwise OR of two operands. See bit(3f).</td>
</tr>
<tr>
<td>perror</td>
<td>Writes system error messages.</td>
</tr>
<tr>
<td>putc</td>
<td>Writes a character to a logical unit.</td>
</tr>
<tr>
<td>qsort</td>
<td>Performs a quick sort of array elements.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rand</td>
<td>Generates random numbers (use random instead).</td>
</tr>
<tr>
<td>random</td>
<td>Generates random numbers.</td>
</tr>
<tr>
<td>rename</td>
<td>Changes the name of a file.</td>
</tr>
<tr>
<td>rshift</td>
<td>Shifts a word right by $n$ bits. See bit(3f).</td>
</tr>
<tr>
<td>shcom_connect</td>
<td>Lets a shared library’s data be shared between call_shared processes.</td>
</tr>
<tr>
<td>short</td>
<td>Converts INTEGER<em>4 to INTEGER</em>2. See long(3f).</td>
</tr>
<tr>
<td>signal</td>
<td>Changes the action for a signal.</td>
</tr>
<tr>
<td>sleep</td>
<td>Suspends execution for an interval.</td>
</tr>
<tr>
<td>srand</td>
<td>Initializes the seed for successive invocations of rand, drand, and irand.</td>
</tr>
<tr>
<td>stat</td>
<td>Returns file status.</td>
</tr>
<tr>
<td>symlink</td>
<td>Creates a symbolic link to an existing file. See link(3f).</td>
</tr>
<tr>
<td>system</td>
<td>Calls system(3).</td>
</tr>
<tr>
<td>time</td>
<td>Returns the system time.</td>
</tr>
<tr>
<td>ttynam</td>
<td>Finds the name of a terminal port.</td>
</tr>
<tr>
<td>umask</td>
<td>Sets a file mode creation mask.</td>
</tr>
<tr>
<td>unlink</td>
<td>Removes a directory entry.</td>
</tr>
<tr>
<td>wait</td>
<td>Waits for a process to terminate.</td>
</tr>
<tr>
<td>xor</td>
<td>Returns the bitwise exclusive OR of two operands. See bit(3f).</td>
</tr>
</tbody>
</table>

1 TU*X only

**Parallel Library Routines (TU*X)**

Compaq Fortran provides library routines you can use for directed parallel decomposition (when specifying the -mp and -omp compiler options) on Tru64 UNIX systems:
When writing new programs that call the run-time parallel library routines, use the OpenMP Fortran API format.

For a helpful program interface, you should add the following statement to the program unit containing the parallel routines:

```fortran
INCLUDE '/usr/include/forompdef.f'
```

For more information on a specific routine, see the appropriate reference page; for example, for more information on `omp_set_lock`, see `omp_set_lock(3f).

**For More Information:**

- On parallel routines, see your user manual.
- On parallel directives, see [Parallel Directives for Tru64 UNIX Systems](#).

### OpenMP Fortran API Run-Time Library Routines (TU*X)

The following table summarizes the Compaq Fortran OpenMP Fortran API run-time library routines. These routines are all external procedures.

**Summary of OpenMP Fortran Parallel Routines (TU*X)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_set_num_threads</code></td>
<td>Sets the number of threads to use for the next parallel region.</td>
</tr>
<tr>
<td><code>omp_get_num_threads</code></td>
<td>Gets the number of threads currently in the team executing the parallel region from which the routine is called.</td>
</tr>
<tr>
<td><code>omp_get_max_threads</code></td>
<td>Gets the maximum value that can be returned by calls to the <code>omp_get_num_threads()</code> function.</td>
</tr>
<tr>
<td><code>omp_get_thread_num</code></td>
<td>Gets the thread number, within the team, in the range from zero to <code>omp_get_num_threads()</code> minus one.</td>
</tr>
<tr>
<td><code>omp_get_num_procs</code></td>
<td>Gets the number of processors that are available to the program.</td>
</tr>
</tbody>
</table>
The following table summarizes the equivalent parallel thread routines you can use for compatibility with existing programs. For new programs, you should use the OpenMP Fortran API Run-Time Library Routines.

For most of the routines, two spellings are shown: an _Otsxxxx form and an mpc_xxxx form. These routine names are equivalent; for example, calling _OtsGetNumThreads is the same as calling mpc_numthreads.

### Parallel Threads Routines for Compatibility (TU*X)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_in_parallel</td>
<td>Informs whether or not a region is executing in parallel.</td>
</tr>
<tr>
<td>omp_set_dynamic</td>
<td>Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.</td>
</tr>
<tr>
<td>omp_get_dynamic</td>
<td>Informs whether or not dynamic thread adjustment is enabled.</td>
</tr>
<tr>
<td>omp_set_nested</td>
<td>Enables or disables nested parallelism.</td>
</tr>
<tr>
<td>omp_get_nested</td>
<td>Informs whether or not nested parallelism is enabled.</td>
</tr>
<tr>
<td>omp_init_lock</td>
<td>Initializes a lock to be used in subsequent calls.</td>
</tr>
<tr>
<td>omp_destroy_lock</td>
<td>Disassociates a lock variable from any locks.</td>
</tr>
<tr>
<td>omp_set_lock</td>
<td>Makes the executing thread wait until the specified lock is available.</td>
</tr>
<tr>
<td>omp_unset_lock</td>
<td>Releases the executing thread from ownership of a lock.</td>
</tr>
<tr>
<td>omp_test_lock</td>
<td>Tries to set the lock associated with a lock variable.</td>
</tr>
</tbody>
</table>

---

**Summary of Parallel Routines for Compatibility (TU*X)**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_OtsInitParallel</td>
<td>Equivalent to the <code>c$OMP PARALLEL</code> directive, which defines the start of a parallel region.</td>
</tr>
<tr>
<td>_OtsStopWorkers</td>
<td>Equivalent to the <code>c$OMP PARALLEL</code> directive, which defines the end of a parallel region.</td>
</tr>
<tr>
<td>mpc_destroy</td>
<td>Equivalent to <code>omp_get_num_threads</code>. Gets the number of threads currently in the team executing the parallel region from which the routine is called.</td>
</tr>
<tr>
<td>_OtsGetNumThreads</td>
<td>Equivalent to <code>omp_get_num_threads</code>. Gets the maximum value that can be returned by calls to the _OtsGetNumThreads() function.</td>
</tr>
<tr>
<td>mpc_numthreads</td>
<td>Equivalent to <code>omp_get_thread_num</code>. Gets the thread number, within the team, in the range from zero to _OtsGetNumThreads() minus one.</td>
</tr>
<tr>
<td>_OtsInParallel</td>
<td>Equivalent to <code>omp_in_parallel</code>. Informs whether or not a region is executing in parallel.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>REAL</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>8</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>4</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER*&lt;n&gt;</td>
<td>n 1</td>
</tr>
</tbody>
</table>
### FORTRAN 77 Intrinsic Functions

<table>
<thead>
<tr>
<th>Function syntax</th>
<th>Type of return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (gen)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>ACOS (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>AIMAG (cmp8)</td>
<td>REAL</td>
</tr>
<tr>
<td>AINT (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>ALOG (real4)</td>
<td>REAL</td>
</tr>
<tr>
<td>ALOG10 (real4)</td>
<td>REAL</td>
</tr>
<tr>
<td>AMAX0 (intA, intB [, intC ] ...)</td>
<td>REAL</td>
</tr>
<tr>
<td>AMAX1 (real4A, real4B, [, real4C ] ...)</td>
<td>REAL</td>
</tr>
<tr>
<td>AMIN0 (intA, intB [, intC ] ...)</td>
<td>REAL</td>
</tr>
<tr>
<td>AMIN1 (real4A, real4B [, real4C ] ...)</td>
<td>REAL</td>
</tr>
<tr>
<td>AMOD (value, mod)</td>
<td>REAL</td>
</tr>
<tr>
<td>ANINT (value)</td>
<td>REAL</td>
</tr>
<tr>
<td>ASIN (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>ATAN (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>ATAN2 (realA, realB)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>CABS (cmp)</td>
<td>Same as argument; COMPLEX returns REAL</td>
</tr>
<tr>
<td>CCOS (cmp8)</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>CHAR (int)</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>CLOG (cmp8)</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>CMPLX (genA [, genB ])</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>CONJG</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>COS</td>
<td>Same as argument</td>
</tr>
<tr>
<td>COSH</td>
<td>Same as argument</td>
</tr>
<tr>
<td>CSIN</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>CSQRT</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>DABS</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DACOS</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DASIN</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DATAN</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DATAN2</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DBLE</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DCOS</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DCOSH</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DDIM</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DEXP</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DIM</td>
<td>Same as arguments</td>
</tr>
<tr>
<td>DINT</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DLOG</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DLOG10</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DMAX1</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DMIN1</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DMOD</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DNINT</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DPROD</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DREAL</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>DSIGN</td>
<td>(dblA, dblB) DOUBLE PRECISION</td>
</tr>
<tr>
<td>DSIN</td>
<td>(dbl) DOUBLE PRECISION</td>
</tr>
<tr>
<td>DSINH</td>
<td>(dbl) DOUBLE PRECISION</td>
</tr>
<tr>
<td>DQURT</td>
<td>(rvalue) DOUBLE PRECISION</td>
</tr>
<tr>
<td>DTAN</td>
<td>(dbl) DOUBLE PRECISION</td>
</tr>
<tr>
<td>DTANH</td>
<td>(dbl) DOUBLE PRECISION</td>
</tr>
<tr>
<td>EXP</td>
<td>(gen) Same as argument</td>
</tr>
<tr>
<td>FLOAT</td>
<td>(ivalue) REAL</td>
</tr>
<tr>
<td>IABS</td>
<td>(int) Same as argument</td>
</tr>
<tr>
<td>ICHAR</td>
<td>(char) INTEGER</td>
</tr>
<tr>
<td>IDIM</td>
<td>(intA, intB) INTEGER</td>
</tr>
<tr>
<td>IDINT</td>
<td>(dbl) INTEGER</td>
</tr>
<tr>
<td>IDNINT</td>
<td>(dbl) INTEGER</td>
</tr>
<tr>
<td>IFIX</td>
<td>(real4) REAL</td>
</tr>
<tr>
<td>INDEX</td>
<td>(charA, charB) INTEGER</td>
</tr>
<tr>
<td>INT</td>
<td>(gen) INTEGER</td>
</tr>
<tr>
<td>ISIGN</td>
<td>(intA, intB) INTEGER</td>
</tr>
<tr>
<td>LEN</td>
<td>(char) INTEGER</td>
</tr>
<tr>
<td>LGE</td>
<td>(charA, charB) LOGICAL</td>
</tr>
<tr>
<td>LGT</td>
<td>(charA, charB) LOGICAL</td>
</tr>
<tr>
<td>LLE</td>
<td>(charA, charB) LOGICAL</td>
</tr>
<tr>
<td>LLT</td>
<td>(charA, charB) LOGICAL</td>
</tr>
<tr>
<td>LOG</td>
<td>(gen) Same as argument</td>
</tr>
<tr>
<td>LOG10</td>
<td>(real) Same as argument</td>
</tr>
<tr>
<td>MAX</td>
<td>(genA, genB [, genC ] ...) INTEGER or REAL</td>
</tr>
</tbody>
</table>
### 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]`

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX0 (intA, intB [, intC ] ...)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MAX1 (realA, realB [, realC ] ...)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MIN (genA, genB [, genC ] ...)</td>
<td>INTEGER or REAL</td>
</tr>
<tr>
<td>MIN0 (intA, intB [, intC ] ...)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MIN1 (realA, real [, real ] ...)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MOD (genA, genB)</td>
<td>REAL</td>
</tr>
<tr>
<td>NINT (real)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL (gen)</td>
<td>REAL</td>
</tr>
<tr>
<td>SIGN (genA, genB)</td>
<td>INTEGER or REAL</td>
</tr>
<tr>
<td>SIN (gen)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>SINH (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>SNGL (dbl)</td>
<td>REAL</td>
</tr>
<tr>
<td>SQRT (gen)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>TAN (real)</td>
<td>Same as argument</td>
</tr>
<tr>
<td>TANH (real)</td>
<td>Same as argument</td>
</tr>
</tbody>
</table>
[, \texttt{STATUS=status}] )

\textbf{COMMON} \left/[\texttt{cname}] / \texttt{nlist}[, ,] / \texttt{cname} / \texttt{nlist} \right] ... 

\textbf{COMPLEX} \texttt{vnam} \left[\left(\texttt{dim}\right)\right] [, \texttt{vname} \left[\left(\texttt{dim}\right)\right]] ... 

\textbf{CONTINUE}

\textbf{DATA} \texttt{nlist} / \texttt{clist}/ \left[, \right] \texttt{nlist} / \texttt{clist}/ \right] ... 

\textbf{DIMENSION} \texttt{array} \left[\left(\texttt{lower}\!:\!\texttt{upper}[, ,] \left(\texttt{lower}\!:\!\texttt{upper}\right)\right)\right] 

\textbf{DO} \left[\texttt{label} [, , ] \right] \texttt{dovar} = \texttt{start}, \texttt{stop} [, , \texttt{inc}]

\textbf{DOUBLE PRECISION} \texttt{vname} \left[\left(\texttt{dim}\right)\right] [, \texttt{vname} \left[\left(\texttt{dim}\right)\right]] ... 

\textbf{ELSE} 
\hspace{1em} \textbf{statementblock}

\textbf{ELSE IF} (\texttt{expression}) \textbf{THEN} 
\hspace{1em} \textbf{statementblock}

\textbf{END}

\textbf{END IF}

\textbf{ENDFILE} \{\texttt{unitspec} | 
\hspace{1em} \texttt{UNIT=unitspec}
\hspace{1em} [, \texttt{ERR=errlabel}]
\hspace{1em} [, \texttt{IOSTAT=iocheck}] \}

\textbf{ENTRY} \texttt{ename} [ ( \left[\texttt{formal} [, , \texttt{formal}] \right] )]

\textbf{EQUIVALENCE} \left(\texttt{nlis}ight)[, \left(\texttt{nlis}\right)] ... 

\textbf{EXTERNAL} \texttt{name} [, \texttt{name}] ... 

\textbf{FORMAT} [\texttt{editlist}]

[ \texttt{type}] \textbf{FUNCTION} \texttt{func} \left(\left[\texttt{formal} [, , \texttt{formal}] \right] \right)

\textbf{GOTO} \texttt{variable} [ [, , \left(\texttt{labels}\right)]]

\textbf{GOTO} \left(\texttt{labels}\right) [, , \texttt{n}]

\textbf{GOTO} \texttt{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]
  [, FORMATTED=formatted] [, IOSTAT=ioccheck]
  [, NAME=name] [, NAMED=named]
  [, NEXTREC=nexrec] [, NUMBER=num] [, OPENED=opened]
  [, RECL=rec] [, 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=ioccheck]
  [, RECL=rec] [, STATUS=status} )

PARAMETER (name=constexpr [, name=constexpr] ...)
\[\text{REAL } \text{vname } [(\text{dim})] \left[, \text{ vname } [(\text{dim})] \right] \ldots\]

\text{RETURN } [\text{ordinal } ]

\text{REWIND } \{ \text{unitspec } | \left[\text{UNIT=} \text{unitspec} \right. \left.\right. \right. \left[, \text{ ERR=} \text{errlabel} \right. \left.\right. \left.\right. \left[, \text{ IOSTAT=} \text{iocheck} \right) \right)\}

\text{SAVE } [\text{names}]$

\text{STOP } [\text{message}]

\text{SUBROUTINE } \text{subr } \left[\left[\text{formal } \left[, \text{ formal } \right. \ldots \right] \right)\]

\text{WRITE } ( \left[\text{UNIT=} \text{unitspec} \right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left[, \text{ FMT=} \text{formatspec} \right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left[, \text{ ERR=} \text{errlabel} \right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left[, \text{ IOSTAT=} \text{iocheck} \right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left.\right. \left[, \text{ REC=} \text{rec} \right) \right) \right) \right) \right)}$

ioolist
Summary of Language Extensions

This appendix summarizes the Compaq Fortran language extensions to the ANSI/ISO Fortran 95 Standard.

For more information, see the following sections:

- Compaq Fortran Language Extensions
- High Performance Fortran Language Extensions

Compaq Fortran Language Extensions

This section summarizes the Compaq Fortran language extensions. Most extensions are available on all systems, but some extensions are limited to certain platforms. If an extension is limited, it is labeled.

Extensions in the following topics are discussed:

- Source Forms
- Names
- Character Sets
- Intrinsic Data Types
- Constants
- Expressions and Assignment
- Specification Statements
- Execution Control
- Compilation Control Statements
- Built-In Functions
- I/O Statements
- I/O Formatting
- File Operation Statements
- Compiler Directives
- Intrinsic Procedures
- Additional Language Features

For information on other extensions, 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, W9*)
- ANSI Character Code Chart (WNT, W9*)
- Key Code Charts (WNT, W9*)

¹ See the printed Compaq Fortran Language Reference Manual.

Intrinsic Data Types

The following are data-type extensions:

<table>
<thead>
<tr>
<th>BYTE</th>
<th>INTEGER*1</th>
<th>REAL*8 ²</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL*1</td>
<td>INTEGER*2</td>
<td>REAL*16 ³</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td>INTEGER*4</td>
<td>COMPLEX*8</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>INTEGER*8 ¹</td>
<td>COMPLEX*16 ²</td>
</tr>
<tr>
<td>LOGICAL*8 ¹</td>
<td>REAL*4</td>
<td>COMPLEX*32 ³</td>
</tr>
</tbody>
</table>

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.

**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

**Specification Statements**

The following specification attributes and statements are extensions:

- AUTOMATIC attribute and statement
- STATIC attribute and statement
- VOLATILE attribute and statement

**Execution Control**

The following control statements are extensions to Fortran 95:

- ASSIGN
- Assigned GO TO
- PAUSE

These are older Fortran features that have been deleted in Fortran 95. Compaq Fortran fully supports these features.

**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)
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
- The `H` edit descriptor

This is an older Fortran feature that has been deleted in Fortran 95. Compaq Fortran fully supports this feature.

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, W9*)
  - BUFFERED
  - BLOCKSIZE
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN', 'BINARY' (WNT, W9*)
  - IOFOCUS (WNT, W9*)
  - KEYED (VMS only)
  - MODE as a synonym for ACTION (WNT, W9*)
- ORGANIZATION
- RECORDTYPE
- SHARE (WNT, W9*)

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, W9*)
  - INITIALSIZE (VMS only)
  - IOFOCUS (WNT, W9*)
  - KEY (VMS only)
  - MAXREC
  - MODE as a synonym for ACTION (WNT, W9*)
  - NAME as a synonym for FILE
  - NOSPANBLOCKS (VMS only)
  - ORGANIZATION
  - READONLY
  - RECORDSIZE as a synonym for RECL
  - RECORDTYPE
  - SHARE (WNT, W9*)
  - SHARED
  - TITLE (WNT, W9*)
  - TYPE as a synonym for STATUS
  - USEROPEN

See also OPEN Statement.

- UNLOCK statement

**Compiler Directives**

The following General Compiler Directives are extensions:

- ALIAS
- ATTRIBUTES
The following OpenMP Fortran API parallel directives are extensions available on Tru64 UNIX systems:

- ATOMIC
- BARRIER
- CRITICAL
- DO
- FLUSH
- MASTER
- ORDERED
- PARALLEL
- PARALLEL DO
- PARALLEL SECTIONS
- SECTIONS
- SINGLE
- THREADPRIVATE

The following Compaq Fortran parallel directives are extensions available on Tru64 UNIX systems:

- BARRIER
- CHUNK
- COPYIN
- CRITICAL SECTION
- INSTANCE
- MP_SCHEDTYPE
- PARALLEL
- PARALLEL DO
PARALLEL SECTIONS
PDO
PDONE
PSECTIONS
SINGLE PROCESS
TASKCOMMON

Intrinsic Procedures

The following intrinsic procedures are extensions available on all platforms:

<table>
<thead>
<tr>
<th>ACOSD</th>
<th>DFLOAT</th>
<th>IISIGN</th>
<th>JMIN0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAX0</td>
<td>DFLOTI</td>
<td>IMAX0</td>
<td>JMIN1</td>
</tr>
<tr>
<td>AIMIN0</td>
<td>DFLOTJ</td>
<td>IMAX1</td>
<td>JMOD</td>
</tr>
<tr>
<td>AJMAX0</td>
<td>DIMAG</td>
<td>IMIN0</td>
<td>JMVBITS</td>
</tr>
<tr>
<td>AJMIN0</td>
<td>DREAL</td>
<td>IMIN1</td>
<td>JNINT</td>
</tr>
<tr>
<td>AND</td>
<td>DSIND</td>
<td>IMOD</td>
<td>JNOT</td>
</tr>
<tr>
<td>ASIND</td>
<td>DTAND</td>
<td>IMVBITS</td>
<td>JZEXT</td>
</tr>
<tr>
<td>ASM</td>
<td>EOF</td>
<td>ININT</td>
<td>LEADZ</td>
</tr>
<tr>
<td>ATAN2D</td>
<td>ERRSNS</td>
<td>INOT</td>
<td>LOC</td>
</tr>
<tr>
<td>ATAND</td>
<td>EXIT</td>
<td>INT1</td>
<td>LSHIFT</td>
</tr>
<tr>
<td>BTEST</td>
<td>FASM</td>
<td>INT2</td>
<td>MALLOC</td>
</tr>
<tr>
<td>BJTEST</td>
<td>FLOATI</td>
<td>INT4</td>
<td>MULT_HIGH</td>
</tr>
<tr>
<td>CDABS</td>
<td>FLOATJ</td>
<td>ISHA</td>
<td>NWORKERS</td>
</tr>
<tr>
<td>CDCOS</td>
<td>FP_CLASS</td>
<td>ISHC</td>
<td>OR</td>
</tr>
<tr>
<td>CDEXP</td>
<td>FREE</td>
<td>ISHL</td>
<td>POPCNT</td>
</tr>
<tr>
<td>CDLOG</td>
<td>HFIX</td>
<td>ISNAN</td>
<td>POPPAR</td>
</tr>
<tr>
<td>CDSIN</td>
<td>IARGPTR</td>
<td>IZEXT</td>
<td>RAN</td>
</tr>
<tr>
<td>CDSQRT</td>
<td>IBCHNG</td>
<td>JFIX</td>
<td>RANDU</td>
</tr>
<tr>
<td>COSD</td>
<td>IDATE</td>
<td>JIAND</td>
<td>RSHIFT</td>
</tr>
</tbody>
</table>
The following intrinsic procedures are extensions available on Alpha processors:

<table>
<thead>
<tr>
<th>COTAN</th>
<th>IIABS</th>
<th>JIBCLR</th>
<th>SECNDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTAND</td>
<td>IIAND</td>
<td>JIBITS</td>
<td>SIND</td>
</tr>
<tr>
<td>DACOSD</td>
<td>IIABS</td>
<td>JIBCLR</td>
<td>SIZEOF</td>
</tr>
<tr>
<td>DASIND</td>
<td>IIABS</td>
<td>JIBCLR</td>
<td>SIND</td>
</tr>
<tr>
<td>DASM</td>
<td>IIABS</td>
<td>JIBCLR</td>
<td>SIND</td>
</tr>
<tr>
<td>DATAN</td>
<td>IIDIM</td>
<td>JIDIM</td>
<td>TAND</td>
</tr>
<tr>
<td>DATAN2</td>
<td>IIDIM</td>
<td>JIDIM</td>
<td>TAND</td>
</tr>
<tr>
<td>DATAN2D</td>
<td>IIDIM</td>
<td>JIDIM</td>
<td>TAND</td>
</tr>
<tr>
<td>DATAND</td>
<td>IIDIM</td>
<td>JIDIM</td>
<td>TAND</td>
</tr>
<tr>
<td>DATE</td>
<td>IIDIM</td>
<td>JIDIM</td>
<td>TAND</td>
</tr>
<tr>
<td>DCMPLEX</td>
<td>IFIX</td>
<td>JISHFT</td>
<td>ZABS</td>
</tr>
<tr>
<td>DCONJG</td>
<td>IINT</td>
<td>JISIGN</td>
<td>ZABS</td>
</tr>
<tr>
<td>DCOSD</td>
<td>IOR</td>
<td>JISIGN</td>
<td>ZABS</td>
</tr>
<tr>
<td>DCOTAN</td>
<td>IISHFT</td>
<td>JMAX0</td>
<td>ZSIN</td>
</tr>
<tr>
<td>DCOTAND</td>
<td>IISHFT</td>
<td>JMAX1</td>
<td>ZSQRT</td>
</tr>
</tbody>
</table>

The following intrinsic procedures are extensions available on OpenVMS, Tru64 UNIX, and Linux systems:

<table>
<thead>
<tr>
<th>AKMAX0</th>
<th>KIBCLR</th>
<th>KINT</th>
<th>KMIN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKMIN0</td>
<td>KIBITS</td>
<td>KIOR</td>
<td>KMOD</td>
</tr>
<tr>
<td>BKTEST</td>
<td>KIBSET</td>
<td>KISHFT</td>
<td>KMVBITS</td>
</tr>
<tr>
<td>DFLOTK</td>
<td>KDIM</td>
<td>KISHFTC</td>
<td>KNINT</td>
</tr>
<tr>
<td>FLOATK</td>
<td>KDINT</td>
<td>KISIGN</td>
<td>KNOT</td>
</tr>
<tr>
<td>INT8</td>
<td>KIDNNT</td>
<td>KMAX0</td>
<td>KZEXT</td>
</tr>
<tr>
<td>KIABS</td>
<td>KIEOR</td>
<td>KMAX1</td>
<td></td>
</tr>
<tr>
<td>KIAND</td>
<td>KIFIX</td>
<td>KMIN0</td>
<td></td>
</tr>
</tbody>
</table>

The following intrinsic procedures are extensions available on OpenVMS, Tru64 UNIX, and Linux systems:
The intrinsic procedure `IARGCOUNT` is an extension available on OpenVMS systems.

## 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)
- `FORTRAN-66 Interpretation` of the `EXTERNAL` statement
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for an I/O record specifier
High Performance Fortran Language Extensions

This section summarizes the High Performance Fortran language extensions to the Fortran 95 standard.

The following extensions are discussed:

- Procedure Prefixes
- Intrinsic Procedures

For information on other extensions, see Compaq Fortran Language Extensions.

**Procedure Prefixes**

The following prefixes are allowed in functions and subroutines as extensions:

- EXTRINSIC (HPF) - functional only on Tru64 UNIX systems
- EXTRINSIC (HPF_LOCAL) - functional only on Tru64 UNIX systems
- EXTRINSIC (HPF_SERIAL) - functional only on Tru64 UNIX systems

**Intrinsic Procedures**

**System Inquiry Intrinsic Procedures**

The following intrinsic procedures are extensions:

- NUMBER_OF_PROCESSORS intrinsic function
- PROCESSORS SHAPE intrinsic function

**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.

Remember that INTEGER(8) is only available on Alpha systems, and REAL(16) and COMPLEX(16) are only available on OpenVMS, Tru64 UNIX, and Linux systems.

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 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 (below sections are in the *Programmer's Guide*):
- 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).
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 \textbf{PRODUCT}(\text{ARRAY[, DIM][, MASK]}), you can skip the argument DIM by using the statement: \texttt{array3} = \texttt{PRODUCT(array1, MASK = array2)}.

The Fortran procedures and statements have been organized into the following tables:

- Program Unit Calls and Definition
- Program Control Statements and Procedures
- Specifying Variables
- System, Drive, and Directory Procedures
- File Management
- Input/Output Procedures
- Random Number Procedures
- Date and Time Procedures
- Keyboard and Speaker Procedures
- Error Handling
- Argument Inquiry
- Memory Allocation and Deallocation Procedures
- Array Procedures
- Numeric and Type Conversion Procedures
- Trigonometric, Exponential, Root, and Logarithmic Procedures
- Floating-Point Inquiry and Control Procedures
- Character Procedures
- Bit Operation and Representation Procedures
- QuickWin Procedures
- Graphics Procedures
- Dialog Procedures
- General Compiler Directives
- National Language Standard Procedures
- Portability Procedures
- COM and Automation Procedures
- Serial Port Procedures
- Miscellaneous Run-Time Procedures (FOR_* routines)
- Functions Not Allowed as Actual Arguments

For more information on keywords, see \textbf{Argument Keywords in Intrinsic Procedures}. For information on using procedures in general, see \textbf{Program Units and Procedures}.  

## Program Unit Calls and Definitions: table

All the following are statements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK DATA</td>
<td>Identifies a block-data subprogram</td>
</tr>
<tr>
<td>CALL</td>
<td>Executes a subroutine</td>
</tr>
<tr>
<td>COMMON</td>
<td>Delineates variables shared between program units</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>Identifies start of a module within a host module</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Specifies a secondary entry point to a subroutine or external function</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Declares a user-defined subroutine or function to be passable as an argument</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>Identifies a program unit as a function</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Inserts the contents of a specified file into the source file</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Specifies an explicit interface for external functions and subroutines</td>
</tr>
<tr>
<td>INTRINSIC</td>
<td>Declares a predefined function</td>
</tr>
<tr>
<td>MODULE</td>
<td>Identifies a module program unit</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Identifies a program unit as a main program</td>
</tr>
<tr>
<td>RETURN</td>
<td>Returns control to the program unit that called a subroutine or function</td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td>Identifies a program unit as a subroutine</td>
</tr>
<tr>
<td>USE</td>
<td>Gives a program unit access to a module</td>
</tr>
</tbody>
</table>

## Program Control Statements and Procedures: table
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CASE</strong></td>
<td>Within a <strong>SELECT CASE</strong> structure, marks a block of statements that are executed if an associated value matches the <strong>SELECT CASE</strong> expression</td>
</tr>
<tr>
<td><strong>CONTINUE</strong></td>
<td>Often used as the target of <strong>GOTO</strong> or as the terminal statement in a <strong>DO</strong> loop; performs no operation</td>
</tr>
<tr>
<td><strong>CYCLE</strong></td>
<td>Advances control to the end statement of a <strong>DO</strong> loop; the intervening loop statements are not executed</td>
</tr>
<tr>
<td><strong>DO</strong></td>
<td>Evaluates statements in the <strong>DO</strong> loop, through and including the ending statement, a specific number of times</td>
</tr>
<tr>
<td><strong>DO WHILE</strong></td>
<td>Evaluates statements in the <strong>DO WHILE</strong> loop, through and including the ending statement, until a logical condition becomes <code>.FALSE.</code></td>
</tr>
<tr>
<td><strong>ELSE</strong></td>
<td>Introduces an <strong>ELSE</strong> block</td>
</tr>
<tr>
<td><strong>ELSE IF</strong></td>
<td>Introduces an <strong>ELSE IF</strong> block</td>
</tr>
<tr>
<td><strong>ELSEWHERE</strong></td>
<td>Introduces an <strong>ELSEWHERE</strong> block</td>
</tr>
<tr>
<td><strong>END</strong></td>
<td>Marks the end of a program unit</td>
</tr>
<tr>
<td><strong>END DO</strong></td>
<td>Marks the end of a series of statements following a <strong>DO</strong> or <strong>DO WHILE</strong> statement</td>
</tr>
<tr>
<td><strong>END FORALL</strong></td>
<td>Marks the end of a series of statements following a block <strong>FORALL</strong> statement</td>
</tr>
<tr>
<td><strong>END IF</strong></td>
<td>Marks the end of a series of statements following a block <strong>IF</strong> statement</td>
</tr>
<tr>
<td><strong>END SELECT</strong></td>
<td>Marks the end of a <strong>SELECT CASE</strong> statement</td>
</tr>
<tr>
<td><strong>END WHERE</strong></td>
<td>Marks the end of a series of statements following a block <strong>WHERE</strong> statement</td>
</tr>
<tr>
<td><strong>EXIT</strong></td>
<td>Leaves a <strong>DO</strong> loop; execution continues with the first statement following</td>
</tr>
<tr>
<td><strong>FORALL</strong></td>
<td>Controls conditional execution of other statements</td>
</tr>
</tbody>
</table>
### Specifying Variables: table

The following are all statements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Declares a variable on the stack, rather than at a static memory location.</td>
</tr>
<tr>
<td>BYTE</td>
<td>Specifies variables as the <code>BYTE</code> data type; <code>BYTE</code> is equivalent to INTEGER(1).</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>Specifies variables as the <code>CHARACTER</code> data type.</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>Specifies variables as the <code>COMPLEX</code> data type.</td>
</tr>
<tr>
<td>DATA</td>
<td>Assigns initial values to variables.</td>
</tr>
<tr>
<td><strong>DIMENSION</strong></td>
<td>Identifies a variable as an array and specifies the number of elements.</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DOUBLE COMPLEX</strong></td>
<td>Specifies variables as the <strong>DOUBLE COMPLEX</strong> data type, equivalent to COMPLEX(8).</td>
</tr>
<tr>
<td><strong>DOUBLE PRECISION</strong></td>
<td>Specifies variables as the <strong>DOUBLE-PRECISION</strong> real data type, equivalent to REAL(8).</td>
</tr>
<tr>
<td><strong>EQUIVALENCE</strong></td>
<td>Specifies that two or more variables or arrays share the same memory location.</td>
</tr>
<tr>
<td><strong>IMPLICIT</strong></td>
<td>Specifies the default typing for real and integer variables and functions.</td>
</tr>
<tr>
<td><strong>INTEGER</strong></td>
<td>Specifies variables as the <strong>INTEGER</strong> data type.</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td>Specifies variables as the <strong>LOGICAL</strong> data type.</td>
</tr>
<tr>
<td><strong>MAP...END MAP</strong></td>
<td>Within a <strong>UNION</strong> statement, delimits a group of variable type declarations that are to be ordered contiguously within memory.</td>
</tr>
<tr>
<td><strong>NAMELIST</strong></td>
<td>Declares a group name for a set of variables to be read or written in a single statement.</td>
</tr>
<tr>
<td><strong>PARAMETER</strong></td>
<td>Equates a constant expression with a name.</td>
</tr>
<tr>
<td><strong>REAL</strong></td>
<td>Specifies variables as the <strong>REAL</strong> data type.</td>
</tr>
<tr>
<td><strong>RECORD</strong></td>
<td>Declares one or more variables of a user-defined structure type.</td>
</tr>
<tr>
<td><strong>SAVE</strong></td>
<td>Causes variables to retain their values between invocations of the procedure in which they are defined.</td>
</tr>
<tr>
<td><strong>STATIC</strong></td>
<td>Declares a variable is in a static memory location, rather than on the stack.</td>
</tr>
<tr>
<td><strong>STRUCTURE...END STRUCTURE</strong></td>
<td>Defines a new variable type, composed of a collection of other variable types.</td>
</tr>
<tr>
<td><strong>TYPE...END TYPE</strong></td>
<td>Defines a new variable type, composed of a collection of other variable types.</td>
</tr>
<tr>
<td><strong>UNION...END UNION</strong></td>
<td>Within a structure, causes two or more maps to occupy the same memory locations.</td>
</tr>
</tbody>
</table>
**System, Drive, and Directory Procedures: table**

All the following are run-time functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGEDIRQQ</td>
<td>CHANGEDIRQQ(dir). Makes the specified directory the current (default) directory.</td>
</tr>
<tr>
<td>CHANGEDRIVEQQ</td>
<td>CHANGEDRIVEQQ(drive). Makes the specified drive the current drive.</td>
</tr>
<tr>
<td>DELDIRQQ</td>
<td>DELDIRQQ(dir). Deletes a specified directory.</td>
</tr>
<tr>
<td>GETDRIVEDIRQQ</td>
<td>GETDRIVEDIRQQ(drivedir). Returns the current drive and directory path.</td>
</tr>
<tr>
<td>GETDRIVESIZEQQ</td>
<td>GETDRIVESIZEQQ(drive, total, avail). Gets the size of the specified drive.</td>
</tr>
<tr>
<td>GETDRIVESQ</td>
<td>GETDRIVESQ(). Reports the drives available to the system.</td>
</tr>
<tr>
<td>GETENVQQ</td>
<td>GETENVQQ(varname, value). Gets a value from the current environment.</td>
</tr>
<tr>
<td>MAKEDIRQQ</td>
<td>MAKEDIRQQ(dirname). Makes a directory with the specified directory name.</td>
</tr>
<tr>
<td>RUNQQ</td>
<td>RUNQQ(filename, commandline). Calls another program and waits for it to execute</td>
</tr>
<tr>
<td>SETENVQQ</td>
<td>SETENVQQ(varvalue). Adds a new environment variable, or sets the value of an existing one.</td>
</tr>
<tr>
<td>SYSTEMQQ</td>
<td>SYSTEMQQ(commandline). Executes a command by passing a command string to the operating system's command interpretor.</td>
</tr>
</tbody>
</table>

**File Management: table**
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELFILESQQ</td>
<td>Run-time Function</td>
<td>DELFILESQQ(files). Deletes the specified files in a specified directory.</td>
</tr>
<tr>
<td>FINDFILEQQ</td>
<td>Run-time Function</td>
<td>FINDFILEQQ(filename, varname, pathbuf). Searches for a file in the directories specified in the PATH environment variable.</td>
</tr>
<tr>
<td>FULLPATHQQ</td>
<td>Run-time Function</td>
<td>FULLPATHQQ(name, pathbuf). Returns the full path for a specified file or directory.</td>
</tr>
<tr>
<td>GETDRIVEDIRQQ</td>
<td>Run-time Function</td>
<td>GETDRIVEDIRQQ(drivedir). Returns current drive and directory path.</td>
</tr>
<tr>
<td>GETFILEINFOQQ</td>
<td>Run-time Function</td>
<td>GETFILEINFOQQ(files, buffer, handle). Returns information about files with names that match a request string.</td>
</tr>
<tr>
<td>PACKTIMEQQ</td>
<td>Run-time Subroutine</td>
<td>PACKTIMEQQ(timedate, iyr, imon, iday, ihr, imin, isec). Packs time values for use by SETFILETIMEQQ.</td>
</tr>
<tr>
<td>RENAMEFILEQQ</td>
<td>Run-time Function</td>
<td>RENAMEFILEQQ(oldname, newname). Renames a file.</td>
</tr>
<tr>
<td>SETFILEACCESSQQ</td>
<td>Run-time Function</td>
<td>SETFILEACCESSQQ(filename, access). Sets file-access mode for the specified file.</td>
</tr>
<tr>
<td>SETFILETIMEQQ</td>
<td>Run-time Function</td>
<td>SETFILETIMEQQ(filename, timedate). Sets modification time for a given file.</td>
</tr>
<tr>
<td>SPLITPATHQQ</td>
<td>Run-time Function</td>
<td>SPLITPATHQQ(path, drive, dir, name, ext). Breaks a full path into four components.</td>
</tr>
<tr>
<td>TRACEBACKQQ</td>
<td>Run-time Subroutine</td>
<td>TRACEBACKQQ(string, user_exit_code, status, expr). Generates a stack trace.</td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>Statement</td>
<td>Similar to a formatted, sequential <strong>READ</strong> statement.</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>Statement</td>
<td>Positions a file to the beginning of the previous record.</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Statement</td>
<td>Disconnects the specified unit.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Statement</td>
<td>Deletes a record from a relative file.</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>Statement</td>
<td>Writes an end-of-file record.</td>
</tr>
<tr>
<td>EOF</td>
<td>Intrinsic Function</td>
<td><strong>EOF</strong>(unit). Checks for end-of-file record.</td>
</tr>
<tr>
<td>INQUIRE</td>
<td>Statement</td>
<td>Returns the properties of a file or unit.</td>
</tr>
<tr>
<td>OPEN</td>
<td>Statement</td>
<td>Associates a unit number with an external device or file.</td>
</tr>
<tr>
<td>PRINT (or TYPE)</td>
<td>Statement</td>
<td>Displays data on the screen.</td>
</tr>
<tr>
<td>READ</td>
<td>Statement</td>
<td>Transfers data from a file to the items in an I/O list.</td>
</tr>
<tr>
<td>REWIND</td>
<td>Statement</td>
<td>Repositions a file to its first record.</td>
</tr>
<tr>
<td>REWRITE</td>
<td>Statement</td>
<td>Rewrites the current record.</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>Statement</td>
<td>Frees a record in a relative or sequential file that was locked by a previous <strong>READ</strong> statement.</td>
</tr>
<tr>
<td>WRITE</td>
<td>Statement</td>
<td>Transfers data from the items in an I/O list to a file</td>
</tr>
</tbody>
</table>
### Random Number Procedures: table

**Note:** Square brackets [...] denote optional arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RAN</strong></td>
<td>Intrinsic function</td>
<td><code>result = RAN(i)</code>. Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1.</td>
</tr>
<tr>
<td><strong>RANDOM</strong></td>
<td>Run-time Subroutine</td>
<td><strong>CALL RANDOM</strong>(ranval). Returns a pseudorandom real value greater than or equal to zero and less than one.</td>
</tr>
<tr>
<td><strong>RANDOM_NUMBER</strong></td>
<td>Intrinsic Subroutine</td>
<td><strong>CALL RANDOM_NUMBER</strong>(harvest). Returns a pseudorandom real value greater than or equal to zero and less than one.</td>
</tr>
<tr>
<td><strong>RANDOM_SEED</strong></td>
<td>Intrinsic Subroutine</td>
<td><strong>CALL RANDOM_SEED</strong>(size [, put] [, get]). Changes the starting point of <strong>RANDOM_NUMBER</strong>; takes one or no arguments.</td>
</tr>
<tr>
<td><strong>RANDU</strong></td>
<td>Intrinsic Subroutine</td>
<td><strong>CALL RANDU</strong>(i1, i2, x). Computes a pseudorandom number as a single-precision value.</td>
</tr>
<tr>
<td><strong>SEED</strong></td>
<td>Run-time Subroutine</td>
<td><strong>CALL SEED</strong>(seedval). Changes the starting point of <strong>RANDOM</strong>.</td>
</tr>
</tbody>
</table>

### Date and Time Procedures: table

**Note:** Square brackets [...] denote optional arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU_TIME</strong></td>
<td>Intrinsic Subroutine</td>
<td><strong>CALL CPU_TIME</strong>(time). Returns the processor time in seconds.</td>
</tr>
</tbody>
</table>
### Keyboard and Speaker Procedures: table

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEEPQQ</td>
<td>Run-time Subroutine</td>
<td>CALL BEEPQQ(freq, duration). Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.</td>
</tr>
<tr>
<td>GETCHARQQ</td>
<td>Run-time Function</td>
<td>GETCHARQQ( ). Returns the next keyboard keystroke.</td>
</tr>
<tr>
<td>GETSTRQQ</td>
<td>Run-time Function</td>
<td>GETSTRQQ(buffer). Reads a character string from the keyboard using buffered input.</td>
</tr>
</tbody>
</table>
### Error Handling: table

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETLASTERRORQQ</td>
<td>Run-time Function</td>
<td>GETLASTERRORQQ( ). Returns the last error set by a run-time function or subroutine.</td>
</tr>
<tr>
<td>MATHERRQQ¹</td>
<td>Run-time Subroutine</td>
<td>CALL MATHERRQQ(name, len, info, retcode). Replaces default error handling for errors from intrinsic math functions.</td>
</tr>
<tr>
<td>SETERRORMODEQQ</td>
<td>Run-time Subroutine</td>
<td>CALL SETERRORMODEQQ(prompt). Sets the mode for handling critical errors.</td>
</tr>
</tbody>
</table>

¹ x86 only

### Argument Inquiry: table

**Note:** Square brackets [...] denote optional arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATED</td>
<td>Intrinsic Function</td>
<td>ALLOCATED(array). Determines whether an allocatable array is allocated.</td>
<td>array: allocatable array&lt;br&gt;result: Logical scalar</td>
</tr>
<tr>
<td>ASSOCIATED</td>
<td>Intrinsic Function</td>
<td>ASSOCIATED(pointer[, target]). Determines whether a pointer and (optional) target are associated.</td>
<td>pointer: any type&lt;br&gt;target: any type&lt;br&gt;result: Logical</td>
</tr>
<tr>
<td>DIGITS</td>
<td>Intrinsic Function</td>
<td>DIGITS(x). Returns number of significant digits for data of the same type as x.</td>
<td>x: Integer or Real&lt;br&gt;result: Integer</td>
</tr>
<tr>
<td><strong>EPSILON</strong></td>
<td>Intrinsic Function</td>
<td><strong>EPSILON</strong>(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.</td>
<td>x: Real result: same type as x</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>GETARG</strong></td>
<td>Run-time Subroutine</td>
<td><strong>CALL GETARG</strong>(n, buffer [, status]). Returns the specified command line argument (where the command itself is argument number zero).</td>
<td>n: INTEGER(2) or INTEGER(4) buffer: Character*(*), status: INTEGER(2)</td>
</tr>
<tr>
<td><strong>HUGE</strong></td>
<td>Intrinsic Function</td>
<td><strong>HUGE</strong>(x). Returns the largest number that can be represented by numbers of type x.</td>
<td>x: Integer or Real result: same type as x</td>
</tr>
<tr>
<td><strong>ILEN</strong></td>
<td>Intrinsic Function</td>
<td><strong>ILEN</strong>(i). Returns the length (in bits) of the two's complement representation of an integer.</td>
<td>i: Integer. result: same type as i</td>
</tr>
<tr>
<td><strong>KIND</strong></td>
<td>Intrinsic Function</td>
<td><strong>KIND</strong>(x). Returns the value of the kind parameter of x.</td>
<td>x: any intrinsic type result: Integer</td>
</tr>
<tr>
<td><strong>LOC</strong></td>
<td>Intrinsic Function</td>
<td><strong>LOC</strong>(a). Returns the address of a. a can be a variable, function call, expression, or constant.</td>
<td>a: any type result: INTEGER(4)</td>
</tr>
<tr>
<td><strong>%LOC</strong></td>
<td>Intrinsic Function</td>
<td>Same as <strong>LOC</strong>.</td>
<td></td>
</tr>
<tr>
<td><strong>MAXEXPONENT</strong></td>
<td>Intrinsic Function</td>
<td><strong>MAXEXPONENT</strong>(x). Returns the largest positive decimal exponent for data of the same type as x.</td>
<td>x: Real result: INTEGER(4)</td>
</tr>
</tbody>
</table>
| **MINEXPONENT** | Intrinsic Function | **MINEXPONENT**(x). Returns the largest negative decimal exponent for data of the same type as x. | x: Real
result: INTEGER(4) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NARGS</strong></td>
<td>Run-time Function</td>
<td><strong>NARGS</strong>(.). Returns the total number of command-line arguments, including the command.</td>
<td>result: INTEGER(4)</td>
</tr>
</tbody>
</table>
| **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** (r, p). 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 |
<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th><strong>Procedure Type</strong></th>
<th><strong>Description</strong></th>
<th><strong>Argument/Function Type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALLOCATE</strong></td>
<td>Statement</td>
<td>Dynamically establishes allocatable array dimensions.</td>
<td></td>
</tr>
<tr>
<td><strong>ALLOCATED</strong></td>
<td>Intrinsic Function</td>
<td>ALLOCATED(array). Determines whether an allocatable array is allocated.</td>
<td>array: allocatable array result: Logical scalar</td>
</tr>
<tr>
<td><strong>DEALLOCATE</strong></td>
<td>Statement</td>
<td>Frees the storage space previously reserved in an ALLOCATE statement.</td>
<td></td>
</tr>
<tr>
<td><strong>FREE</strong></td>
<td>Intrinsic Subroutine</td>
<td>FREE(addr). Frees the memory block specified by the integer pointer addr.</td>
<td>addr: INTEGER(4)</td>
</tr>
<tr>
<td><strong>MALLOC</strong></td>
<td>Intrinsic Function</td>
<td>MALLOC(size). Allocates a memory block of size size bytes and returns an integer pointer to the block.</td>
<td>size: INTEGER(4) result: INTEGER(4)</td>
</tr>
</tbody>
</table>

**Array Procedures: table**

**Note:** Square brackets [...] denote optional arguments.
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
</table>
| **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 |
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CSHIFT</strong></td>
<td>Intrinsic Function</td>
<td><code>CSHIFT(array, shift [, dim])</code>. Performs a circular shift along (optional) dimension <code>dim</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>array</code>: any type</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>shift</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dim</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>result: same type and shape as <code>array</code></td>
</tr>
<tr>
<td><strong>DIMENSION</strong></td>
<td>Statement</td>
<td>Identifies a variable as an array and specifies the number of elements.</td>
</tr>
<tr>
<td><strong>DOT_PRODUCT</strong></td>
<td>Intrinsic Function</td>
<td><code>DOT_PRODUCT(vector_a, vector_b)</code>. Performs dot-product multiplication on vectors (one-dimensional arrays).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>vector_a</code>: any except Character</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>vector_b</code>: same type and size as <code>vector_a</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>result: a scalar of the same type as <code>vector_a</code></td>
</tr>
<tr>
<td><strong>EOSHIFT</strong></td>
<td>Intrinsic Function</td>
<td><code>EOSHIFT(array, shift [, boundary] [, dim])</code>. Shifts elements off one end of <code>array</code> along (optional) dimension <code>dim</code> and copies (optional) <code>boundary</code> values in other end.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>array</code>: any type</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>shift</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>boundary</code>: same as <code>array</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dim</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>result: same type and shape as <code>array</code></td>
</tr>
<tr>
<td><strong>LBOUND</strong></td>
<td>Intrinsic Function</td>
<td><code>LBOUND(array [, dim] [, kind])</code>. Returns lower dimensional bound (s) of an array along dimension <code>dim</code> (optional).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>array</code>: any type</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dim</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>kind</code>: Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>result: Integer and a scalar if <code>dim</code> is absent or <code>array</code> is one-dimensional; otherwise, a vector</td>
</tr>
<tr>
<td>Intrinsic Function</td>
<td>Function Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| **MATMUL**        | **MATMUL**(*matrix_a, matrix_b*). Performs matrix multiplication on matrices (twodimensional arrays). | *matrix_a*: any except Character  
*matrix_b*: same as  
*matrix_a*  
result: same type as  
*matrix_a* |                                                     |
| **MAXLOC**        | **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**        | **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**         | **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* |                                                     |
<table>
<thead>
<tr>
<th>Function</th>
<th>Intrinsic Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
</table>
| MINLOC | `MINLOC(array [, dim] [, mask])`. | Returns the location of the minimum value in an array meeting conditions in (optional) mask along optional dimension dim. | array: Integer or Real  
dim: Integer  
mask: Logical | Integer vector whose size is equal to the number of dimensions in array |
| MINVAL | `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 | Same type as array and a scalar if dim is absent or array is one-dimensional; otherwise, one dimension smaller than array |
| PACK | `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 | A vector (one-dimensional array) of the same type as array |
| PRODUCT | `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 | Same type as array and a scalar if dim is absent or array is one-dimensional; otherwise, one |
<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESHAPE</strong></td>
<td><strong>RESHAPE</strong>(source, shape [, pad] [, order]). Reshapes an array with subscript order (optional), padded with array elements pad (optional).</td>
<td>source: any type, shape: Integer, pad: same as source, order: Integer</td>
<td>result: same type as source and same shape as shape</td>
</tr>
<tr>
<td><strong>SHAPE</strong></td>
<td><strong>SHAPE</strong>(source [, kind]). Returns the shape of an array.</td>
<td>source: any type, kind: Integer</td>
<td>result: a vector (one-dimensional array) of the same type as source</td>
</tr>
<tr>
<td><strong>SIZE</strong></td>
<td><strong>SIZE</strong>(array [, dim] [, kind]). Returns the extent of array along dimension dim (optional).</td>
<td>array: any type, dim: Integer, kind: Integer</td>
<td>result: Integer scalar</td>
</tr>
<tr>
<td><strong>SORTQQ</strong></td>
<td><strong>CALL SORTQQ</strong>(addr, count, size). Sorts a one-dimensional array of non-structure data types (derived types are not allowed).</td>
<td>addr: INTEGER(4), count: INTEGER(4), size: INTEGER(4)</td>
<td></td>
</tr>
<tr>
<td><strong>SPREAD</strong></td>
<td><strong>SPREAD</strong>(source, dim, ncopies). Replicates an array by adding a dimension.</td>
<td>source: any type, dim: Integer, ncopies: Integer</td>
<td>result: same type as source and one dimension larger</td>
</tr>
</tbody>
</table>
| Intrinsic Function | **SUM**(*array [, dim] [, 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* |  
| Intrinsic Function | **TRANSPOSE**(*matrix*) | **matrix**: any type  
result: a two-dimensional array the same type as *matrix* |  
| Intrinsic Function | **UBOUND**(*array [, dim] [, kind]*) | **array**: any type  
**dim**: Integer  
**kind**: Integer  
result: same type as *array* and a scalar if **dim** is absent or *array* is one-dimensional; otherwise, a vector. |  
| Intrinsic Function | **UNPACK**(*vector, mask, 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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Intrinsic Function</td>
<td>ABS(a). Returns absolute value of a. When ABS is passed as an argument, a must be REAL(4).</td>
<td>$a$: Integer, Real or Complex result: same type as a, except Real for Complex</td>
</tr>
<tr>
<td>AIMAG</td>
<td>Intrinsic Function</td>
<td>AIMAG (z). Returns imaginary part of complex number z.</td>
<td>$z$: COMPLEX(4) result: REAL(4)</td>
</tr>
<tr>
<td>AINT</td>
<td>Intrinsic Function</td>
<td>AINT (a[, kind]). Truncates a to whole number of specified kind (optional). When AINT is passed as an argument, a must be REAL(4).</td>
<td>$a$: Real kind: Integer result: Real of type kind if present; else same as a</td>
</tr>
<tr>
<td>AMAX0</td>
<td>Intrinsic Function</td>
<td>AMAX0(a1, a2 [, a3...]). Returns largest value among integer arguments as real.</td>
<td>All $a$: INTEGER(4) result: REAL(4)</td>
</tr>
<tr>
<td>AMIN0</td>
<td>Intrinsic Function</td>
<td>AMIN0(a1, a2 [, a3...]). Returns smallest value among integer arguments as real.</td>
<td>All $a$: INTEGER(4) result: REAL(4)</td>
</tr>
<tr>
<td>ANINT</td>
<td>Intrinsic Function</td>
<td>ANINT(a [, kind]). Rounds to nearest whole number of specified kind (optional). When ANINT is passed as an argument, a must be REAL(4).</td>
<td>$a$: Real kind: Integer result: Real of type kind if present; else same as a</td>
</tr>
<tr>
<td>CEILING</td>
<td>Intrinsic Function</td>
<td>CEILING(a [, kind]). Returns smallest integer greater than a.</td>
<td>$a$: Real kind: Integer result: INTEGER(4)</td>
</tr>
<tr>
<td>Function</td>
<td>Intrinsic Function</td>
<td>Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| CMPLX      |                    | **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      |                    | **CONJG***(z)*. Returns the conjugate of a complex number.                   | *z*: COMPLEX(4)  
result: COMPLEX(4)                                                                                       |
| DBLE       |                    | **DBLE***(a)*. Converts *a* to double precision type.                        | *a*: Integer, Real, or Complex.  
result: REAL(8)                                                                                             |
| DCMPLX     |                    | **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     |                    | **DFLOAT***(a)*. Converts an integer to double precision type.               | *a*: Integer  
result: REAL(8)                                                                                           |
| DIM        |                    | **DIM***(x, y)*. Returns *x*-y if positive; else 0.  
When **DIM** is passed as an argument, *a* must be REAL(4).                                                      | *x*: Integer or Real  
*y*: same as *x*  
result: same type as *x*                                                                                     |
| DPROD      |                    | **DPROD***(x, y)*. Returns double-precision product of single precision *x* and *y*. | *x*: REAL(4)  
*y*: REAL (4)  
result: REAL(8)                                                                                           |
<table>
<thead>
<tr>
<th>Function</th>
<th>Intrinsic Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FLOAT</strong></td>
<td>Intrinsic Function</td>
<td>FLOAT(i). Converts i to REAL(4).</td>
<td>i: Integer</td>
<td>result: REAL(4)</td>
</tr>
<tr>
<td><strong>FLOOR</strong></td>
<td>Intrinsic Function</td>
<td>FLOOR(a [, kind]). Returns the greatest integer less than or equal to a.</td>
<td>a: Real, kind: Integer</td>
<td>result: Integer</td>
</tr>
<tr>
<td><strong>IFIX</strong></td>
<td>Intrinsic Function</td>
<td>IFIX(a). Converts a single-precision real argument to an integer argument by truncating.</td>
<td>x: REAL(4)</td>
<td>result: Default integer (usually INTEGER(4))</td>
</tr>
<tr>
<td><strong>IMAG</strong></td>
<td>Intrinsic Function</td>
<td>Same as AIMAG.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INT</strong></td>
<td>Intrinsic Function</td>
<td>INT(a [, kind]). Converts a value to integer type.</td>
<td>a: Integer, Real, or Complex, kind: Integer</td>
<td>result: Integer of type kind if present; else same as a</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td>Intrinsic Function</td>
<td>LOGICAL(l [, kind]). Converts between logical arguments of (optional) kind.</td>
<td>l: Logical, kind: Integer</td>
<td></td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>Intrinsic Function</td>
<td>MAX(a1, a2 [, a3...]). Returns largest value among arguments.</td>
<td>All a: any type</td>
<td>result: same type as a</td>
</tr>
<tr>
<td><strong>MAX1</strong></td>
<td>Intrinsic Function</td>
<td>MAX1(a1, a2 [, a3...]). Returns largest value among real arguments as integer.</td>
<td>All a: Real(4)</td>
<td>result: Integer</td>
</tr>
<tr>
<td><strong>MIN</strong></td>
<td>Intrinsic Function</td>
<td>MIN(a1, a2 [, a3...]). Returns largest value among arguments.</td>
<td>All a: any type</td>
<td>result: same type as a</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
<td>Parameters</td>
<td>Result Type</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>MIN1</td>
<td>Intrinsic</td>
<td>Returns smallest value among real arguments as integer</td>
<td>All a: REAL(4)</td>
<td>result: Integer</td>
</tr>
<tr>
<td>MOD</td>
<td>Intrinsic</td>
<td>Returns remainder of (a/p). When MOD is passed as an argument, (a) must be integer.</td>
<td>a: Integer or Real</td>
<td>p: same as a</td>
</tr>
<tr>
<td>MODULO</td>
<td>Intrinsic</td>
<td>Returns (a \mod p).</td>
<td>a: Integer or Real</td>
<td>p: same as a</td>
</tr>
<tr>
<td>NINT</td>
<td>Intrinsic</td>
<td>Returns the nearest integer to (a).</td>
<td>a: Real</td>
<td>kind: Integer</td>
</tr>
<tr>
<td>REAL</td>
<td>Intrinsic</td>
<td>Converts a value to real type.</td>
<td>a: Integer, Real, or Complex</td>
<td>result: REAL(4)</td>
</tr>
<tr>
<td>SIGN</td>
<td>Intrinsic</td>
<td>Returns absolute value of (a) times the sign of (b).</td>
<td>a: Integer or Real</td>
<td>b: same as a</td>
</tr>
<tr>
<td>SNGL</td>
<td>Intrinsic</td>
<td>Converts a double-precision argument to single-precision real type.</td>
<td>a: REAL(8)</td>
<td>result: REAL(4)</td>
</tr>
</tbody>
</table>
### Trigonometric, Exponential, Root, and Logarithmic Procedures: table

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACOS</strong></td>
<td>( \text{ACOS}(x) ). Returns the arc cosine of ( x ) in radians between 0 and pi. When <strong>ACOS</strong> is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real &lt;br&gt;result: same type as ( x )</td>
</tr>
<tr>
<td><strong>ACOSD</strong></td>
<td>( \text{ACOSD}(x) ). Returns the arc cosine of ( x ) in degrees between 0 and 180. When <strong>ACOSD</strong> is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real &lt;br&gt;result: same type as ( x )</td>
</tr>
<tr>
<td><strong>ALOG</strong></td>
<td>( \text{ALOG}(x) ). Returns natural log of ( x ).</td>
<td>( x ): REAL(4) &lt;br&gt;result: REAL(4)</td>
</tr>
<tr>
<td><strong>ALOG10</strong></td>
<td>( \text{ALOG10}(x) ). Returns common log (base 10) of ( x ).</td>
<td>( x ): REAL(4) &lt;br&gt;result: REAL(4)</td>
</tr>
<tr>
<td><strong>ASIN</strong></td>
<td>( \text{ASIN}(x) ). Returns arc sine of ( x ) in radians between ( \pm \pi/2 ). When <strong>ASIN</strong> is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real &lt;br&gt;result: same type as ( x )</td>
</tr>
<tr>
<td><strong>ASIND</strong></td>
<td>( \text{ASIND}(x) ). Returns arc sine of ( x ) in degrees between ( \pm 90^\circ ). When <strong>ASIND</strong> is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real &lt;br&gt;result: same type as ( x )</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Arguments</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>ATAN</td>
<td>Returns arc tangent of ( x ) in radians between ( \pm \pi/2 ). When ATAN is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real</td>
</tr>
<tr>
<td>ATAND</td>
<td>Returns arc tangent of ( x ) in degrees between ( \pm 90^\circ ). When ATAND is passed as an argument, ( x ) must be REAL(4).</td>
<td>( x ): Real</td>
</tr>
<tr>
<td>ATAN2</td>
<td>Returns the arc tangent of ( y/x ) in radians between ( \pm \pi ). When ATAN2 is passed as an argument, ( y ) and ( x ) must be REAL(4).</td>
<td>( y ): Real, ( x ): same as ( y )</td>
</tr>
<tr>
<td>ATAN2D</td>
<td>Returns the arc tangent of ( y/x ) in degrees between ( \pm 180^\circ ). When ATAN2D is passed as an argument, ( y ) and ( x ) must be REAL(4).</td>
<td>( y ): Real, ( x ): same as ( y )</td>
</tr>
<tr>
<td>CCOS</td>
<td>Returns complex cosine of ( x ).</td>
<td>( x ): COMPLEX(4)</td>
</tr>
<tr>
<td>CDCOS</td>
<td>Returns double-precision complex cosine of ( x ).</td>
<td>( x ): COMPLEX(8)</td>
</tr>
<tr>
<td>CDEXP</td>
<td>Returns double-precision complex value of ( e^{**x} ).</td>
<td>( x ): COMPLEX(8)</td>
</tr>
<tr>
<td>CDLOG</td>
<td>Returns double-precision complex natural log of ( x ).</td>
<td>( x ): COMPLEX(8)</td>
</tr>
<tr>
<td>CDSIN</td>
<td>Returns double-precision complex sine of ( x ).</td>
<td>( x ): COMPLEX(8)</td>
</tr>
<tr>
<td>CDSQRT</td>
<td>Returns double-precision complex square root of ( x ).</td>
<td>( x ): COMPLEX(8)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Arguments</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>CEXP</td>
<td>$CEXP(x)$. Returns complex value of $e^{\ast x}$.</td>
<td>$x$: COMPLEX(4)</td>
</tr>
<tr>
<td>CLOG</td>
<td>$CLOG(x)$. Returns complex natural log of $x$.</td>
<td>$x$: COMPLEX(4)</td>
</tr>
<tr>
<td>COS</td>
<td>$COS(x)$. Returns cosine of $x$ radians. When $COS$ is passed as an argument, $x$ must be REAL(4).</td>
<td>$x$: Real or Complex</td>
</tr>
<tr>
<td>COSD</td>
<td>$COSD(x)$. Returns cosine of $x$ degrees. When $COSD$ is passed as an argument, $x$ must be REAL(4).</td>
<td>$x$: Real</td>
</tr>
<tr>
<td>COSH</td>
<td>$COSH(x)$. Returns the hyperbolic cosine of $x$. When $COSH$ is passed as an argument, $x$ must be REAL(4).</td>
<td>$x$: Real</td>
</tr>
<tr>
<td>COTAN</td>
<td>$COTAN(x)$. Returns cotangent of $x$ in radians.</td>
<td>$x$: Real</td>
</tr>
<tr>
<td>COTAND</td>
<td>$COTAND(x)$. Returns cotangent of $x$ in degrees.</td>
<td>$x$: Real</td>
</tr>
<tr>
<td>CSIN</td>
<td>$CSIN(x)$. Returns complex sine of $x$.</td>
<td>$x$: COMPLEX(4)</td>
</tr>
<tr>
<td>CSQRT</td>
<td>$CSQRT(x)$. Returns complex square root of $x$.</td>
<td>$x$: COMPLEX(4)</td>
</tr>
<tr>
<td>DACOS</td>
<td>$DACOS(x)$. Returns double-precision arc cosine of $x$ in radians between 0 and pi.</td>
<td>$x$: REAL(8)</td>
</tr>
<tr>
<td>DACOSD</td>
<td>$DACOSD(x)$. Returns the arc cosine of $x$ in degrees between 0 and 180. When $DACOSD$ is passed as an argument, $x$ must be REAL(4).</td>
<td>$x$: REAL(8)</td>
</tr>
<tr>
<td>DASIN</td>
<td>$DASIN(x)$. Returns double-precision arc sine of $x$ in radians between $\pm\pi/2$.</td>
<td>$x$: REAL(8)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>DASIND</td>
<td>Returns double-precision arc sine of x in degrees between ±90°.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DATAN</td>
<td>Returns double-precision arc tangent of x in radians between ±pi/2.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DATAND</td>
<td>Returns double-precision arc tangent of x in degrees between ±90°.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DATAN2</td>
<td>Returns double-precision arc tangent of y/x in radians between ±pi.</td>
<td>y: REAL(8)</td>
</tr>
<tr>
<td>DATAN2D</td>
<td>Returns double-precision arc tangent of y/x in degrees between ±180°.</td>
<td>y: REAL(8)</td>
</tr>
<tr>
<td>DCOS</td>
<td>Returns double-precision cosine of x in radians.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DCOSD</td>
<td>Returns double-precision cosine of x in degrees.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DCOSH</td>
<td>Returns double-precision hyperbolic cosine of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DCOTAN</td>
<td>Returns double-precision cotangent of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DEXP</td>
<td>Returns double-precision value of $e^{x}$</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>DLOG</td>
<td>Returns double-precision natural log of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Domain</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>DLOG10</strong></td>
<td>DLOG10(x). Returns double-precision common log (base 10) of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DSIN</strong></td>
<td>DSIN(x). Returns double-precision sin of x in radians.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DSIND</strong></td>
<td>DSIND(x). Returns double-precision sin of x in degrees.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DSINH</strong></td>
<td>DSINH(x). Returns double-precision hyperbolic sine of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DSQRT</strong></td>
<td>DSQRT(x). Returns double-precision square root of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DTAN</strong></td>
<td>DTAN(x). Returns double-precision tangent of x in radians.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DTAND</strong></td>
<td>DTAND(x). Returns double-precision tangent of x in degrees.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>DTANH</strong></td>
<td>DTANH(x). Returns double-precision hyperbolic tangent of x.</td>
<td>x: REAL(8)</td>
</tr>
<tr>
<td><strong>EXP</strong></td>
<td>EXP(x). Returns value of e**x. When EXP is passed as an argument, x must be REAL(4).</td>
<td>x: Real or Complex</td>
</tr>
<tr>
<td><strong>LOG</strong></td>
<td>LOG(x) Returns the natural log of x.</td>
<td>x: Real or Complex</td>
</tr>
<tr>
<td><strong>LOG10</strong></td>
<td>LOG10(x). Returns the common log (base 10) of x.</td>
<td>x: Real</td>
</tr>
<tr>
<td><strong>SIN</strong></td>
<td>SIN(x). Returns the sine of x radians. When SIN is passed as an argument, x must be REAL(4).</td>
<td>x: Real or Complex</td>
</tr>
</tbody>
</table>
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](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS</td>
<td>Intrinsic Function</td>
<td>DIGITS(x). Returns number of significant digits for data of the same type as x.</td>
<td>x: Integer or Real result: Integer</td>
</tr>
<tr>
<td>EPSILON</td>
<td>Intrinsic Function</td>
<td>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.</td>
<td>x: Real result: same type as x</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
<td>Parameters/Result</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td><strong>EXPONENT</strong></td>
<td>Intrinsic</td>
<td><strong>EXPONENT(x)</strong>. Returns the exponent part of the representation of x.</td>
<td>x: Real, result: Integer</td>
</tr>
<tr>
<td><strong>FRACTION</strong></td>
<td>Intrinsic</td>
<td><strong>FRACTION(x)</strong>. Returns the fractional part of the representation of x.</td>
<td>x: Real, result: same type as x</td>
</tr>
<tr>
<td><strong>GETCONTROLFPQQ</strong></td>
<td>Run-time</td>
<td><strong>GETCONTROLFPQQ</strong>(control). Returns the value of the floating-point processor control word.</td>
<td>control: INTEGER(2)</td>
</tr>
<tr>
<td><strong>GETSTATUSFPQQ</strong></td>
<td>Run-time</td>
<td><strong>GETSTATUSFPQQ</strong>(status). Returns the value of the floating-point processor status word.</td>
<td>status: INTEGER(2)</td>
</tr>
<tr>
<td><strong>HUGE</strong></td>
<td>Intrinsic</td>
<td><strong>HUGE(x)</strong>. Returns largest number that can be represented by data of type x.</td>
<td>x: Integer or Real, result: same type as x</td>
</tr>
<tr>
<td><strong>LCWRQQ</strong></td>
<td>Run-time</td>
<td>Same as <strong>SETCONTROLFPQQ</strong>.</td>
<td></td>
</tr>
<tr>
<td><strong>MAXEXPONENT</strong></td>
<td>Intrinsic</td>
<td><strong>MAXEXPONENT(x)</strong>. Returns the largest positive decimal exponent for data of the same type as x.</td>
<td>x: Real, result: INTEGER(4)</td>
</tr>
<tr>
<td><strong>MINEXPONENT</strong></td>
<td>Intrinsic</td>
<td><strong>MINEXPONENT(x)</strong>. Returns the largest negative decimal exponent for data of the same type as x.</td>
<td>x: Real, result: INTEGER(4)</td>
</tr>
</tbody>
</table>
| **NEAREST** | Intrinsic Function | **NEAREST**(x, s). Returns the nearest different machine representable number to x in the direction of the sign of s. | x: Real  
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 |
<p>| <strong>SCWRQQ</strong> | Run-time Subroutine | Same as <strong>GETCONTROLFPQQ</strong>. |  |
| <strong>SETCONTROLFPQQ</strong> | Run-time Subroutine | <strong>SETCONTROLFPQQ</strong> <em>(controlword)</em>. Sets the value of the floating-point processor control word. | controlword: INTEGER(2) |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ACHAR</code></td>
<td>Intrinsic Function</td>
<td><code>ACHAR(i)</code>. Returns character in position <code>i</code> in the ASCII character set.</td>
<td><code>i</code>: Integer from 0 to 255 result: CHARACTER(1)</td>
</tr>
<tr>
<td><code>ADJUSTL</code></td>
<td>Intrinsic Function</td>
<td><code>ADJUSTL(string)</code>. Adjusts left, removing leading blanks and inserting trailing blanks.</td>
<td><code>string</code>: Character*(*) result: same type as <code>string</code></td>
</tr>
<tr>
<td><code>ADJUSTR</code></td>
<td>Intrinsic Function</td>
<td><code>ADJUSTR(string)</code>. Adjusts right, removing trailing blanks and inserting leading blanks.</td>
<td><code>string</code>: Character*(*) result: same type as <code>string</code></td>
</tr>
</tbody>
</table>

1 x86 only

See also [Miscellaneous Run-Time Procedures: table](#).

**Character Procedures: table**

**Note:** Square brackets [...] denote optional arguments.
<table>
<thead>
<tr>
<th>Function</th>
<th>Intrinsic Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| CHAR     | CHAR(i [, kind]). | Returns character in position i in the processor's character set of (optional) kind. | i: Integer  
kind: Integer  
result: CHARACTER(1) of type kind if present; otherwise, default kind. |
| IACHAR   | IACHAR(c).        | Returns the position of c in the ASCII character set.                       | c: CHARACTER(1)  
result: Integer |
| ICHAR    | ICHAR(c).         | Returns the position of c in the processor's character set.                 | c: CHARACTER(1)  
result: Integer |
| INDEX    | INDEX(string, substring, back)]. | Returns the starting position of a substring in a string, leftmost or (optional) rightmost occurrence. | string: Character*(*):  
substring: Character*(*):  
back: Logical  
result: Integer |
| LEN      | LEN(string [, kind]). | Returns the size of the variable string.                                    | string: Character*(*):  
kind: Integer  
result: Integer of type kind if present; otherwise, default kind. |
| LEN_TRIM | LEN_TRIM(string). | Returns the number of characters in string, not counting trailing blanks.   | string: Character*(*)  
result: Integer |
| LGE      | LGE(string_a, string_b). | Tests whether string_a is greater than or equal to string_b, based on the ASCII collating sequence; TRUE if equal or string_a is last. | string_a: Character*(*):  
string_b: Character*(*)  
result: Logical |
<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGT</td>
<td>(LGTF(string_a, string_b)). Tests whether (string_a) is greater than (string_b), based on the ASCII collating sequence; TRUE if (string_a) is last.</td>
<td>(string_a): Character*(<em>), (string_b): Character</em>(*)</td>
<td>Logical</td>
</tr>
<tr>
<td>LLE</td>
<td>(LLEF(string_a, string_b)). Tests whether (string_a) is less than or equal to (string_b), based on the ASCII collating sequence; TRUE if equal or (string_a) is first.</td>
<td>(string_a): Character*(<em>), (string_b): Character</em>(*)</td>
<td>Logical</td>
</tr>
<tr>
<td>LLT</td>
<td>(LLTF(string_a, string_b)). Tests whether (string_a) is less than (string_b), based on the ASCII collating sequence; TRUE if (string_a) is first.</td>
<td>(string_a): Character*(<em>), (string_b): Character</em>(*)</td>
<td>Logical</td>
</tr>
<tr>
<td>REPEAT</td>
<td>(REPEATF(string, ncopies)). Concatenates multiple copies of a string.</td>
<td>(string): Character*(*), (ncopies): Integer</td>
<td>Character*(*)</td>
</tr>
<tr>
<td>SCAN</td>
<td>(SCANF(string, set[, back])). Scans a string for any characters in a set and returns leftmost or (optional) rightmost position where a match is found.</td>
<td>(string): Character*(<em>), (set): Character</em>(*), (back): Logical</td>
<td>Integer</td>
</tr>
<tr>
<td>TRIM</td>
<td>(TRIMF(string)). Removes trailing blanks from a string.</td>
<td>(string): Character*(*)</td>
<td>Character*(*)</td>
</tr>
<tr>
<td>VERIFY</td>
<td>(VERIFYF(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</td>
<td>(string): Character*(<em>), (set): Character</em>(*), (back): Logical</td>
<td>Integer</td>
</tr>
</tbody>
</table>
## Bit Operation and Representation Procedures: table

**Note:** Square brackets [...] denote optional arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BIT SIZE</strong></td>
<td>Intrinsic Function</td>
<td><strong>BIT_SIZE</strong>(i). Returns the number of bits in integers of type i.</td>
<td>i: Integer result: same type as i</td>
</tr>
<tr>
<td><strong>BTEST</strong></td>
<td>Intrinsic Function</td>
<td><strong>BTEST</strong>(i, pos). Tests a bit in position pos of i; true if bit is 1.</td>
<td>i: Integer pos: positive Integer result: Logical</td>
</tr>
<tr>
<td><strong>IAND</strong></td>
<td>Intrinsic Function</td>
<td><strong>IAND</strong>(i, j). Performs a logical AND.</td>
<td>i: Integer j: Integer result: same type as i</td>
</tr>
<tr>
<td><strong>IBCHNG</strong></td>
<td>Intrinsic Function</td>
<td><strong>IBCHNG</strong>(i, pos). Reverses value of bit in position pos of i.</td>
<td>i: Integer pos: positive Integer result: same type as i</td>
</tr>
<tr>
<td><strong>IBCLR</strong></td>
<td>Intrinsic Function</td>
<td><strong>IBCLR</strong>(i, pos). Clears the bit in position pos of i to zero.</td>
<td>i: Integer pos: positive Integer result: same type as i</td>
</tr>
<tr>
<td><strong>IBITS</strong></td>
<td>Intrinsic Function</td>
<td><strong>IBITS</strong>(i, pos, len). Extracts a sequence of bits of length len from i starting in position pos.</td>
<td>i: Integer pos: positive Integer len: positive Integer result: same type as i</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>IBSET</strong></td>
<td><code>IBSET(i, pos)</code>. Sets the bit in position <code>pos</code> of <code>i</code> to one.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>pos</code>: positive Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>IEOR</strong></td>
<td><code>IEOR(i, j)</code>. Performs an exclusive OR.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>j</code>: Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>IOR</strong></td>
<td><code>IOR(i, j)</code>. Performs an inclusive OR.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>j</code>: Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>ISHA</strong></td>
<td><code>ISHA(i, shift)</code>. Shifts <code>i</code> arithmetically left or right by <code>shift</code> bits; left if <code>shift</code> positive, right if <code>shift</code> negative. Zeros shifted in from the right, ones shifted in from the left.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>shift</code>: Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>ISHC</strong></td>
<td><code>ISHC(i, shift)</code>. Performs a circular shift of <code>i</code> left or right by <code>shift</code> bits; left if <code>shift</code> positive, right if <code>shift</code> negative. No bits lost.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>shift</code>: Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>ISHFT</strong></td>
<td><code>ISHFT(i, shift)</code>. Shifts <code>i</code> logically left or right by <code>shift</code> bits; left if <code>shift</code> positive, right if <code>shift</code> negative. Zeros shifted in from opposite end.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>shift</code>: Integer</td>
<td>same type as <code>i</code></td>
</tr>
<tr>
<td><strong>ISHFTC</strong></td>
<td><code>ISHFTC(i, shift[, size])</code>. Performs a circular shift of the rightmost bits of (optional) <code>size</code> by <code>shift</code> bits. No bits lost.</td>
<td><code>i</code>: Integer&lt;br&gt;<code>shift</code>: Integer&lt;br&gt;<code>size</code>: positive Integer</td>
<td>same type as <code>i</code></td>
</tr>
</tbody>
</table>
| **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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABOUTBOXQQ</td>
<td>Function</td>
<td>Adds an About Box with customized text.</td>
</tr>
<tr>
<td>APPENDMENUQQ</td>
<td>Function</td>
<td>Appends a menu item.</td>
</tr>
<tr>
<td>CLICKMENUPQQ</td>
<td>Function</td>
<td>Sends menu click messages to the application window.</td>
</tr>
<tr>
<td>DELETEMENUQQ</td>
<td>Function</td>
<td>Deletes a menu item.</td>
</tr>
<tr>
<td>FOCUSQQ</td>
<td>Function</td>
<td>Makes a child window active, and gives focus to the child window.</td>
</tr>
<tr>
<td>GETACTIVEQQ</td>
<td>Function</td>
<td>Gets the unit number of the active child window.</td>
</tr>
<tr>
<td>GETHWNDQQ</td>
<td>Function</td>
<td>Gets the true windows handle from window with the specified unit number.</td>
</tr>
<tr>
<td>GETWINDOWCONFIG</td>
<td>Function</td>
<td>Returns the current window's properties.</td>
</tr>
<tr>
<td>GETWSIZEQQ</td>
<td>Function</td>
<td>Gets the size of the child or frame window.</td>
</tr>
<tr>
<td>GETUNITQQ</td>
<td>Function</td>
<td>Gets the unit number corresponding to the specified windows handle. Inverse of GETHWNDQQ.</td>
</tr>
<tr>
<td>INCHARQQ</td>
<td>Function</td>
<td>Reads a keyboard input and return its ASCII value.</td>
</tr>
<tr>
<td>INITIALSETTINGS</td>
<td>Function</td>
<td>Controls initial menu settings and initial frame window settings.</td>
</tr>
<tr>
<td>INQFOCUSQQ</td>
<td>Function</td>
<td>Determines which window is active and has the focus.</td>
</tr>
<tr>
<td>INSERTMENUQQ</td>
<td>Function</td>
<td>Inserts a menu item.</td>
</tr>
<tr>
<td>INTEGERTORGB</td>
<td>Subroutine</td>
<td>Converts a true color value into its red, green and blue components.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>MESSAGEBOXQQ</td>
<td>Displays a message box.</td>
<td></td>
</tr>
<tr>
<td>MODIFYMENULEGSQQ</td>
<td>Modifies a menu item state.</td>
<td></td>
</tr>
<tr>
<td>MODIFYMENUROUTINEQQ</td>
<td>Modifies a menu item's callback routine.</td>
<td></td>
</tr>
<tr>
<td>MODIFYMENUSTRINGQQ</td>
<td>Changes a menu item's text string.</td>
<td></td>
</tr>
<tr>
<td>PASSEDIRKEYSQQ</td>
<td>Determines the behavior of direction and page keys.</td>
<td></td>
</tr>
<tr>
<td>REGISTERMOUSEEVENT</td>
<td>Registers the application-defined routines to be called on mouse events.</td>
<td></td>
</tr>
<tr>
<td>RGBTOINTEGER</td>
<td>Converts a trio of red, green and blue values to a true color value for use with RGB functions and subroutines.</td>
<td></td>
</tr>
<tr>
<td>SETACTIVEQQ</td>
<td>Makes the specified window the current active window without giving it the focus.</td>
<td></td>
</tr>
<tr>
<td>SETMESSAGEQQ</td>
<td>Changes any QuickWin message, including status bar messages, state messages and dialog box messages.</td>
<td></td>
</tr>
<tr>
<td>SETWINDOWCONFIG</td>
<td>Configures the current window's properties.</td>
<td></td>
</tr>
<tr>
<td>SETWINDOWMENUQQ</td>
<td>Sets the Window menu to which current child window names will be appended.</td>
<td></td>
</tr>
<tr>
<td>SETWSIZEQQ</td>
<td>Sets the size of the child or frame window.</td>
<td></td>
</tr>
<tr>
<td>UNREGISTERMOUSEEVENT</td>
<td>Removes the callback routine registered by REGISTERMOUSEEVENT.</td>
<td></td>
</tr>
<tr>
<td>WAITONMOUSEEVENT</td>
<td>Blocks return until a mouse event occurs.</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARC, ARC_W</td>
<td>Functions</td>
<td>Draws an arc.</td>
</tr>
<tr>
<td>CLEARSCREEN</td>
<td>Subroutine</td>
<td>Clears the screen, viewport, or text window.</td>
</tr>
<tr>
<td>DISPLAYCURSOR</td>
<td>Function</td>
<td>Turns the cursor off and on.</td>
</tr>
<tr>
<td>ELLIPSE, ELLIPSE_W</td>
<td>Functions</td>
<td>Draws an ellipse or circle.</td>
</tr>
<tr>
<td>FLOODFILL, FLOODFILL_W</td>
<td>Functions</td>
<td>Fills an enclosed area of the screen with the current color index, using the current fill mask.</td>
</tr>
<tr>
<td>FLOODFILLRGB, FLOODFILLRGB_W</td>
<td>Functions</td>
<td>Fills an enclosed area of the screen with the current RGB color, using the current fill mask.</td>
</tr>
<tr>
<td>GETARCINFO</td>
<td>Function</td>
<td>Determines the end points of the most recently drawn arc or pie.</td>
</tr>
<tr>
<td>GETBKCOLOR</td>
<td>Function</td>
<td>Returns the current background color index.</td>
</tr>
<tr>
<td>GETBKCOLORRGB</td>
<td>Function</td>
<td>Returns the current background RGB color.</td>
</tr>
<tr>
<td>GETCOLOR</td>
<td>Function</td>
<td>Returns the current color index.</td>
</tr>
<tr>
<td>GETCOLORRGB</td>
<td>Function</td>
<td>Returns the current RGB color.</td>
</tr>
<tr>
<td>GETCURRENTPOSITION,</td>
<td>Subroutines</td>
<td>Returns the coordinates of the current graphics-output position.</td>
</tr>
<tr>
<td>GETCURRENTPOSITION_W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function/Subroutine</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GETFILLMASK</td>
<td>Subroutine</td>
<td>Returns the current fill mask.</td>
</tr>
<tr>
<td>GETFONTINFO</td>
<td>Function</td>
<td>Returns the current font characteristics.</td>
</tr>
<tr>
<td>GETGTEXTEXTENT</td>
<td>Function</td>
<td>Determines the width of the specified text in the current font.</td>
</tr>
<tr>
<td>GETGTEXTROTATION</td>
<td>Function</td>
<td>Get the current text rotation angle.</td>
</tr>
<tr>
<td>GETIMAGE, GETIMAGE_W</td>
<td>Subroutines</td>
<td>Stores a screen image in memory.</td>
</tr>
<tr>
<td>GETLINESTYLE</td>
<td>Function</td>
<td>Returns the current line style.</td>
</tr>
<tr>
<td>GETPHYSCOORD</td>
<td>Subroutine</td>
<td>Converts viewport coordinates to physical coordinates.</td>
</tr>
<tr>
<td>GETPIXEL, GETPIXEL_W</td>
<td>Functions</td>
<td>Returns a pixel's color index.</td>
</tr>
<tr>
<td>GETPIXELRGB, GETPIXELRGB_W</td>
<td>Functions</td>
<td>Returns a pixel's RGB color.</td>
</tr>
<tr>
<td>GETPIXELS</td>
<td>Function</td>
<td>Returns the color indices of multiple pixels.</td>
</tr>
<tr>
<td>GETPIXELSRGB</td>
<td>Function</td>
<td>Returns the RGB colors of multiple pixels.</td>
</tr>
<tr>
<td>GETTEXTCOLOR</td>
<td>Function</td>
<td>Returns the current text color index.</td>
</tr>
<tr>
<td>GETTEXTCOLOURRGB</td>
<td>Function</td>
<td>Returns the current text RGB color.</td>
</tr>
<tr>
<td>GETTEXTOPTION</td>
<td>Subroutine</td>
<td>Returns the current text-output position.</td>
</tr>
<tr>
<td>GETTEXTWINDOW</td>
<td>Subroutine</td>
<td>Returns the boundaries of the current text window.</td>
</tr>
<tr>
<td>GETVIEWCOORD, GETVIEWCOORD_W</td>
<td>Subroutines</td>
<td>Converts physical or window coordinates to viewport coordinates.</td>
</tr>
<tr>
<td>Function/Subroutine</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GETWINDOWCOORD</td>
<td>Subrtn</td>
<td>Converts viewport coordinates to window coordinates.</td>
</tr>
<tr>
<td>GETWRITEMODE</td>
<td>Function</td>
<td>Returns the logical write mode for lines.</td>
</tr>
<tr>
<td>GRSTATUS</td>
<td>Function</td>
<td>Returns the status (success or failure) of the most recently called graphics routine.</td>
</tr>
<tr>
<td>IMAGESIZE, IMAGESIZE_W</td>
<td>Functions</td>
<td>Returns image size in bytes.</td>
</tr>
<tr>
<td>INITIALIZEFONTS</td>
<td>Function</td>
<td>Initializes the font library.</td>
</tr>
<tr>
<td>LINETO, LINETO_W</td>
<td>Functions</td>
<td>Draws a line from the current position to a specified point.</td>
</tr>
<tr>
<td>LINETOAR</td>
<td>Function</td>
<td>Draws a line between points in one array and corresponding points in another array.</td>
</tr>
<tr>
<td>LINETOAREX</td>
<td>Function</td>
<td>Similar to LINETOAR, but also lets you specify color and line style.</td>
</tr>
<tr>
<td>LOADIMAGE, LOADIMAGE_W</td>
<td>Functions</td>
<td>Reads a Windows bitmap file (.BMP) and displays it at the specified location.</td>
</tr>
<tr>
<td>MOVETO, MOVETO_W</td>
<td>Subrtns</td>
<td>Moves the current position to the specified point.</td>
</tr>
<tr>
<td>OUTGTEXT</td>
<td>Subrtn</td>
<td>Sends text in the current font to the screen at the current position.</td>
</tr>
<tr>
<td>OUTTEXT</td>
<td>Subrtn</td>
<td>Sends text to the screen at the current position.</td>
</tr>
<tr>
<td>PIE, PIE_W</td>
<td>Functions</td>
<td>Draws a pie slice.</td>
</tr>
<tr>
<td>POLYGON, POLYGON_W</td>
<td>Functions</td>
<td>Draws a polygon.</td>
</tr>
<tr>
<td>POLYLINEQQ</td>
<td>Function</td>
<td>Draws a line between successive points in an array.</td>
</tr>
<tr>
<td><strong>PUTIMAGE, PUTIMAGE_W</strong></td>
<td>Subroutines</td>
<td>Retrieves an image from memory and displays it.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td><strong>RECTANGLE, RECTANGLE_W</strong></td>
<td>Functions</td>
<td>Draws a rectangle.</td>
</tr>
<tr>
<td><strong>REMAPALLPALETTERGB</strong></td>
<td>Function</td>
<td>Remaps a set of RGB color values to indices recognized by the current video configuration.</td>
</tr>
<tr>
<td><strong>REMAPPALETTERGB</strong></td>
<td>Function</td>
<td>Remaps a single RGB color value to a color index.</td>
</tr>
<tr>
<td><strong>SAVEIMAGE, SAVEIMAGE_W</strong></td>
<td>Functions</td>
<td>Captures a screen image and saves it as a Windows bitmap file.</td>
</tr>
<tr>
<td><strong>SCROLLTEXTWINDOW</strong></td>
<td>Subroutine</td>
<td>Scrolls the contents of a text window.</td>
</tr>
<tr>
<td><strong>SETBKCOLOR</strong></td>
<td>Function</td>
<td>Sets the current background color.</td>
</tr>
<tr>
<td><strong>SETBKCOLORRGB</strong></td>
<td>Function</td>
<td>Sets the current background color to a direct color value rather than an index to a defined palette.</td>
</tr>
<tr>
<td><strong>SETCLIPRGN</strong></td>
<td>Subroutine</td>
<td>Limits graphics output to a part of the screen.</td>
</tr>
<tr>
<td><strong>SETCOLOR</strong></td>
<td>Function</td>
<td>Sets the current color to a new color index.</td>
</tr>
<tr>
<td><strong>SETCOLORRGB</strong></td>
<td>Function</td>
<td>Sets the current color to a direct color value rather than an index to a defined palette.</td>
</tr>
<tr>
<td><strong>SETFILLMASK</strong></td>
<td>Subroutine</td>
<td>Changes the current fill mask to a new pattern.</td>
</tr>
<tr>
<td><strong>SETFONT</strong></td>
<td>Function</td>
<td>Finds a single font matching the specified characteristics and assigns it to <strong>OUTGTEXT</strong>.</td>
</tr>
<tr>
<td><strong>SETGTEXTROTATION</strong></td>
<td>Subroutine</td>
<td>Sets the direction in which text is written to the specified angle.</td>
</tr>
<tr>
<td>Function/Subroutine</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SETLINESTYLE</td>
<td>Subroutine</td>
<td>Changes the current line style.</td>
</tr>
<tr>
<td>SETMOUSECURSOR</td>
<td>Function</td>
<td>Sets the mouse cursor for the window in focus.</td>
</tr>
<tr>
<td>SETPIXEL, SETPIXEL_W</td>
<td>Functions</td>
<td>Sets color of a pixel at a specified location.</td>
</tr>
<tr>
<td>SETPIXELRGB, SETPIXELRGB_W</td>
<td>Functions</td>
<td>Sets RGB color of a pixel at a specified location.</td>
</tr>
<tr>
<td>SETPIXELS</td>
<td>Subroutine</td>
<td>Sets the color indices of multiple pixels.</td>
</tr>
<tr>
<td>SETPIXELSRGB</td>
<td>Subroutine</td>
<td>Sets the RGB color of multiple pixels.</td>
</tr>
<tr>
<td>SETTEXTCOLOR</td>
<td>Function</td>
<td>Sets the current text color to a new color index.</td>
</tr>
<tr>
<td>SETTEXTCOLORRGB</td>
<td>Function</td>
<td>Sets the current text color to a direct color value rather than an index to a defined palette.</td>
</tr>
<tr>
<td>SETTEXTCURSOR</td>
<td>Function</td>
<td>Sets the height and width of the text cursor for the window in focus.</td>
</tr>
<tr>
<td>SETTEXTPosition</td>
<td>Subroutine</td>
<td>Changes the current text position.</td>
</tr>
<tr>
<td>SETTEXTWINDOW</td>
<td>Subroutine</td>
<td>Sets the current text display window.</td>
</tr>
<tr>
<td>SETVIEWORG</td>
<td>Subroutine</td>
<td>Positions the viewport coordinate origin.</td>
</tr>
<tr>
<td>SETVIEWPORT</td>
<td>Subroutine</td>
<td>Defines the size and screen position of the viewport.</td>
</tr>
<tr>
<td>SETWINDOW</td>
<td>Function</td>
<td>Defines the window coordinate system.</td>
</tr>
<tr>
<td>SETWRITEMODE</td>
<td>Function</td>
<td>Changes the current logical write mode for lines.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLGEXIT</strong></td>
<td>Subroutine</td>
<td><strong>CALL DLGEXIT(dlg)</strong>. Closes an open dialog.</td>
</tr>
<tr>
<td><strong>DLGFLUSH</strong></td>
<td>Subroutine</td>
<td><strong>CALL DLGFLUSH(dlg [,flushall])</strong>. Updates the display of a dialog box.</td>
</tr>
<tr>
<td><strong>DLGGET</strong></td>
<td>Function</td>
<td><strong>DLGGET(dlg, controlid, value [, index])</strong>. Retrieves values of dialog control variables.</td>
</tr>
<tr>
<td><strong>DLGGETCHAR</strong></td>
<td>Function</td>
<td><strong>DLGGETCHAR(dlg, controlid, value [, index])</strong>. Retrieves values of dialog control variables of type Character.</td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>DLGGETINT</td>
<td>DLGGETINT(dlg, controlid, value index]). Retrieves values of dialog control variables of type Integer.</td>
<td></td>
</tr>
<tr>
<td>DLGGETLOG</td>
<td>DLGGETLOG(dlg, controlid, value index]). Retrieves values of dialog control variables of type Logical.</td>
<td></td>
</tr>
<tr>
<td>DLGINIT</td>
<td>DLGINIT(dlgid, dlg). Initializes a dialog.</td>
<td></td>
</tr>
<tr>
<td>DLGINITWITHRESOURCEHANDLE</td>
<td>DLGINITWITHRESOURCEHANDLE (dlgid, hinst, dlg). Initializes a dialog.</td>
<td></td>
</tr>
<tr>
<td>DLGISDLGMESSAGE</td>
<td>DLGISDLGMESSAGE(mesg). Determines whether a message is intended for a modeless dialog box and, if it is, processes it.</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>DLGISDLGMESSAGEWITHDLG</strong></td>
<td>Determines whether a message is intended for a specific modeless dialog box and, if it is, processes it.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGMODAL</strong></td>
<td>Displays a dialog and processes dialog selections from user.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGMODALWITHPARENT</strong></td>
<td>Displays a dialog in a specific parent window and processes dialog selections from user.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGMODELESS</strong></td>
<td>Displays a modeless dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGSENDCTRLMESSAGE</strong></td>
<td>Sends a message to a dialog box control.</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><strong>DLGSET</strong></td>
<td>( \text{DLGSET}(\text{dlg}, \text{controlid}, \text{value} [, \text{index}]) ). Assigns values to dialog control variables.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGSETCHAR</strong></td>
<td>( \text{DLGSETCHAR}(\text{dlg}, \text{controlid}, \text{value} [, \text{index}]) ). Assigns values to dialog control variables of type Character.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGSETCTRLEVENTHANDLER</strong></td>
<td>( \text{DLGSETCTRLEVENTHANDLER}(\text{controlid}, \text{handler}, \text{dispid} [, \text{iid}]) ). Assigns your own event handlers to ActiveX™ controls in a dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>DLGSETINT</strong></td>
<td>( \text{DLGSETINT}(\text{dlg}, \text{controlid}, \text{value} \text{ index}) ). Assigns values to dialog control variables of type Integer.</td>
<td></td>
</tr>
<tr>
<td>Macro/Subroutine</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DLGSETLOG</td>
<td>Function</td>
<td>DLGSETLOG(dlg, controlid, value index]). Assigns values to dialog control variables of type Logical.</td>
</tr>
<tr>
<td>DLGSETRETURN</td>
<td>Subroutine</td>
<td>CALL DLGSETRETURN(dlg, retval) Sets the return value for DLGMODAL.</td>
</tr>
<tr>
<td>DLGSETSUB</td>
<td>Function</td>
<td>DLGSETSUB(dlg, controlid, value index]). Assigns procedures (callback routines) to dialog controls.</td>
</tr>
<tr>
<td>DLGSETTITLE</td>
<td>Subroutine</td>
<td>CALL DLGSETTITLE(dlg, title). Sets the title of a dialog box.</td>
</tr>
<tr>
<td>DLGUNINIT</td>
<td>Subroutine</td>
<td>CALL DLGUNINIT(dlg). Deallocates memory occupied by an initialized dialog.</td>
</tr>
</tbody>
</table>

For more information on using these procedures, see [Using Dialogs](#).

**General Compiler Directives: table**

**Note:** Each directive name is preceded by the prefix `cDEC$`; for example, `cDEC$ ALIAS`. The c can be a c, C, !, or *.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIAS</td>
<td>Specifies an alternate external name to be used when referring to external subprograms.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>Applies attributes to variables and procedures.</td>
</tr>
<tr>
<td>DECLARE</td>
<td>Generates warning messages for undeclared variables.</td>
</tr>
<tr>
<td>DEFINE</td>
<td>Creates a variable whose existence can be tested during conditional compilation.</td>
</tr>
<tr>
<td>ELSE</td>
<td>Marks the beginning of an alternative conditional-compilation block to an <strong>IF</strong> directive construct.</td>
</tr>
<tr>
<td>ELSEIF</td>
<td>Marks the beginning of an alternative conditional-compilation block to an <strong>IF</strong> directive construct.</td>
</tr>
<tr>
<td>ENDIF</td>
<td>Marks the end of a conditional-compilation block.</td>
</tr>
<tr>
<td>FIXEDFORMLINESIZE</td>
<td>Sets fixed-form line length. This directive has no effect on freeform code.</td>
</tr>
<tr>
<td>FREEFORM</td>
<td>Uses freeform format for source code.</td>
</tr>
<tr>
<td>IDENT</td>
<td>Specifies an identifier for an object module.</td>
</tr>
<tr>
<td>IF</td>
<td>Marks the beginning of a conditional-compilation block.</td>
</tr>
<tr>
<td>IF DEFINED</td>
<td>Marks the beginning of a conditional-compilation block.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Selects default integer size.</td>
</tr>
<tr>
<td>IVDEP</td>
<td>Assists the compiler's dependence analysis.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>Sends a character string to the standard output device.</td>
</tr>
<tr>
<td>NODECLARE</td>
<td>(Default) Turns off warning messages for undeclared variables.</td>
</tr>
<tr>
<td>NOFREEFORM</td>
<td>(Default) Uses standard FORTRAN 77 code formatting column rules.</td>
</tr>
<tr>
<td>NOSTRICT</td>
<td>(Default) Disables a previous <strong>STRICT</strong> directive.</td>
</tr>
<tr>
<td>OBJCOMMENT</td>
<td>Specifies a library search path in an object file.</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries.</td>
</tr>
<tr>
<td>PACK</td>
<td>Specifies the memory starting addresses of derived-type items.</td>
</tr>
<tr>
<td>PSECT</td>
<td>Modifies certain characteristics of a common block.</td>
</tr>
<tr>
<td>REAL</td>
<td>Selects default real size.</td>
</tr>
<tr>
<td>STRICT</td>
<td>Disables Visual Fortran features not in the Fortran 90 Standard.</td>
</tr>
<tr>
<td>SUBTITLE</td>
<td>Prints the specified subtitle on subsequent pages of the source listing.</td>
</tr>
<tr>
<td>TITLE</td>
<td>Prints the specified title on subsequent pages of the source listing.</td>
</tr>
<tr>
<td>UNDEFINE</td>
<td>Removes a symbolic variable name created with the DEFINE directive.</td>
</tr>
<tr>
<td>UNROLL</td>
<td>Tells the compiler's optimizer how many times to unroll a DO loop.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBCharLen</td>
<td>Function</td>
<td>Returns the length of the first multibyte character in a string.</td>
</tr>
<tr>
<td>MBConvertMBToUnicode</td>
<td>Function</td>
<td>Converts a character string from a multibyte codepage to a Unicode string.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>MBConvertUnicodeToMB</strong></td>
<td>Converts a Unicode string to a multibyte character string of the current codepage.</td>
<td></td>
</tr>
<tr>
<td><strong>MBCurMax</strong></td>
<td>Returns the longest possible multibyte character for the current codepage.</td>
<td></td>
</tr>
<tr>
<td><strong>MBINCHARQQ</strong></td>
<td>Same as <strong>INCHARQQ</strong>, but can read a single multibyte character at once.</td>
<td></td>
</tr>
<tr>
<td><strong>MBINDEX</strong></td>
<td>Same as <strong>INDEX</strong>, except that multibyte characters can be included in its arguments.</td>
<td></td>
</tr>
<tr>
<td><strong>MBJISToJMS</strong></td>
<td>Converts a Japan Industry Standard (JIS) character to a Kanji (Shift JIS or JMS) character.</td>
<td></td>
</tr>
<tr>
<td><strong>MBJMSToJIS</strong></td>
<td>Converts a Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.</td>
<td></td>
</tr>
<tr>
<td><strong>MBLead</strong></td>
<td>Determines whether a given character is the first byte of a multibyte character.</td>
<td></td>
</tr>
<tr>
<td><strong>MBLen</strong></td>
<td>Returns the number of multibyte characters in a string, including trailing spaces.</td>
<td></td>
</tr>
<tr>
<td><strong>MBLen_Trim</strong></td>
<td>Returns the number of multibyte characters in a string, not including trailing spaces.</td>
<td></td>
</tr>
<tr>
<td><strong>MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE</strong></td>
<td>Same as <strong>LGE, LGT, LLE, LLT</strong>, and the logical operators .EQ. and .NE., except that multibyte characters can be included in their arguments.</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>MBNext</td>
<td>Returns the string position of the first byte of the multibyte character immediately after the given string position.</td>
<td></td>
</tr>
<tr>
<td>MBPrev</td>
<td>Returns the string position of the first byte of the multibyte character immediately before the given string position.</td>
<td></td>
</tr>
<tr>
<td>MBSCAN</td>
<td>Same as SCAN, except that multibyte characters can be included in its arguments.</td>
<td></td>
</tr>
<tr>
<td>MBStrLead</td>
<td>Performs a context sensitive test to determine whether a given byte in a character string is a lead byte.</td>
<td></td>
</tr>
<tr>
<td>MBVERIFY</td>
<td>Same as VERIFY, except that multibyte characters can be included in its arguments.</td>
<td></td>
</tr>
<tr>
<td>NLSEnumCodepages</td>
<td>Returns an array of valid codepages for the current console.</td>
<td></td>
</tr>
<tr>
<td>NLSEnumLocales</td>
<td>Returns an array of locales (language/country combinations) installed on the system.</td>
<td></td>
</tr>
<tr>
<td>NLSFormatCurrency</td>
<td>Formats a currency number according to conventions of the current locale (language/country).</td>
<td></td>
</tr>
<tr>
<td>NLSFormatDate</td>
<td>Formats a date according to conventions of the current locale (language/country).</td>
<td></td>
</tr>
<tr>
<td>NLSFormatNumber</td>
<td>Formats a number according to conventions of the current locale (language/country).</td>
<td></td>
</tr>
<tr>
<td>NLSFormatTime</td>
<td>Formats a time according to conventions of the current locale (language/country).</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
<th>Argument/Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABORT</strong></td>
<td>Subroutine</td>
<td><strong>CALL ABORT([ string]).</strong> Flushes and closes all I/O buffers and terminates execution with the optional abort message in string.</td>
<td><em>string</em>: CHARACTER*(*</td>
</tr>
<tr>
<td><strong>ACCESS</strong></td>
<td>Function</td>
<td><strong>ACCESS</strong>(name, mode). Checks the file specified by name for accessibility to the caller in mode mode.</td>
<td><em>name</em>: CHARACTER*(*</td>
</tr>
</tbody>
</table>
| **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) and **BESYN**(n, x) take arguments integer order n and value x. | n: INTEGER(4)  
x: REAL(4)  
result: REAL(4) |
| **BIC, BIS, BIT** | Subroutines and Function | Bit clear and set subroutines, and bit test function.  
**CALL BIC**(bitnum, target) clears a bit.  
**CALL BIS**(bitnum, target) sets a bit.  
**BIT**(bitnum, source) tests a bit. | bitnum: INTEGER(4)  
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) |
<p>| <strong>CLOCK</strong> | Function | <strong>CLOCK</strong>( ). Returns the time in HH:MM:SS format. | result: CHARACTER(8) |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters/Return Values</th>
</tr>
</thead>
</table>
| **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*) and **DBESYN**(*n*, *x*) take arguments integer order *n* and value *x*. |
| **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** | *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** | *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) |
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDATE</td>
<td>Subroutine</td>
<td>CALL FDATE(string) or FDATE(). Returns the date and time as an ASCII string.</td>
<td>string: CHARACTER*(*)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return from FDATE function: CHARACTER (24)</td>
<td></td>
</tr>
<tr>
<td>FGETC</td>
<td>Function</td>
<td>FGETC(lunit, char). Reads the next available character from lunit and places it in char.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>char: CHARACTER(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>FLUSH</td>
<td>Subroutine</td>
<td>CALL FLUSH(lunit). Causes the contents of the lunit buffer to be flushed to the associated file.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td>FPUTC</td>
<td>Function</td>
<td>FPUTC(lunit, char). Writes a character char to the file associated with lunit, bypassing normal Fortran I/O.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>char: CHARACTER(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>FSEEK</td>
<td>Subroutine</td>
<td>FSEEK(lunit, offset, from). Repositions a file on a logical unit to offset bytes relative to position from.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>offset: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>from: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>FSTAT</td>
<td>Function</td>
<td>FSTAT(lunit, statb). Returns information about lunit in the array statb.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>statb(12): INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>FTELL</td>
<td>Function</td>
<td>FTELL(lunit). Returns the current position of the file associated with lunit as an offset in bytes from the beginning of the file.</td>
<td>lunit: INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>Function/Subroutine</td>
<td>Type</td>
<td>Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>GERROR</td>
<td>Subroutine</td>
<td>CALL GERROR(string). Fills string with a message for the last detected IERRNO error.</td>
<td>string: CHARACTER*(*)</td>
</tr>
<tr>
<td>GETC</td>
<td>Function</td>
<td>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.</td>
<td>char: CHARACTER(1) result: INTEGER(4)</td>
</tr>
<tr>
<td>GETCWD</td>
<td>Function</td>
<td>GETCWD(dirname). Places the current working directory path in dirname.</td>
<td>dirname: CHARACTER*(*) result: INTEGER(4)</td>
</tr>
<tr>
<td>GETENV</td>
<td>Function</td>
<td>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.</td>
<td>ename: CHARACTER*(<em>) evalue: CHARACTER</em>(*)</td>
</tr>
<tr>
<td>GETGID</td>
<td>Function</td>
<td>GETGID( ). Included for portability. Always returns 1.</td>
<td>result: INTEGER(4) equal to 1</td>
</tr>
<tr>
<td>GETLOG</td>
<td>Subroutine</td>
<td>CALL GETLOG(name). Returns the user's login name or blanks.</td>
<td>name: CHARACTER*(*)</td>
</tr>
<tr>
<td>GETPID</td>
<td>Function</td>
<td>GETPID( ). Returns the process ID number of the current process.</td>
<td>result: INTEGER(4)</td>
</tr>
<tr>
<td>GETUID</td>
<td>Function</td>
<td>GETUID( ). Included for portability. Always returns 1.</td>
<td>result: INTEGER(4) equal to 1</td>
</tr>
<tr>
<td>Subroutine/Function</td>
<td>Description</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td><strong>GMTIME</strong></td>
<td>Subroutine</td>
<td><strong>CALL GMTIME(stime, tarray)</strong>. Separates the time returned by <strong>TIME()</strong> in <em>stime</em> into GMT date and time and stores in <em>tarray</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>stime</strong>: INTEGER(4)</td>
<td><strong>tarray(9)</strong>: INTEGER(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HOSTNAM</strong></td>
<td>Function</td>
<td><strong>HOSTNAM(name)</strong>. Puts the name of the current host into <em>name</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>name</strong>: CHARACTER*(*)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IARGC</strong></td>
<td>Function</td>
<td><strong>IARGC()</strong>. Returns the index of the last command-line argument.</td>
<td></td>
</tr>
<tr>
<td><strong>result</strong>: INTEGER(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IDATE</strong></td>
<td>Subroutine</td>
<td><strong>CALL IDATE(iarray)</strong> or <strong>CALL IDATE(month, day, year)</strong>. Returns the day, month and year in <em>iarray</em> or the parameters <em>month</em>, <em>day</em>, and <em>year</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>iarray(3)</strong>: INTEGER(4)</td>
<td><strong>month</strong>: INTEGER(4)</td>
<td><strong>day</strong>: INTEGER(4)</td>
<td><strong>year</strong>: INTEGER(4)</td>
</tr>
<tr>
<td><strong>IERRNO</strong></td>
<td>Function</td>
<td><strong>IERRNO()</strong>. Returns the last IERRNO error code.</td>
<td></td>
</tr>
<tr>
<td><strong>result</strong>: INTEGER(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IRAND, IRANDM</strong></td>
<td>Functions</td>
<td><strong>IRAND(iflag)</strong> and <strong>IRANDM(iflag)</strong>. Return integer random numbers between 0 and ((2^{31}) -1), chosen according to the value of <em>iflag</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>iflag</strong>: INTEGER(4)</td>
<td><strong>result</strong>: INTEGER(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ITIME</strong></td>
<td>Subroutine</td>
<td><strong>CALL ITIME(iarray)</strong>. Returns the current time in <em>iarray</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>iarray(3)</strong>: INTEGER(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>JDATE</strong></td>
<td>Function</td>
<td><strong>JDATE()</strong>. Return the Julian date in the form <strong>YYDDD</strong>.</td>
<td></td>
</tr>
<tr>
<td><strong>result</strong>: CHARACTER(8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Arguments</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| **KILL** | **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** | **LNBLNK**(string). Returns the index of the last nonblank character in string. | string: CHARACTER*(* | result: INTEGER(4) |
| **LONG** | **LONG**(int2). Returns an INTEGER(2) argument as an INTEGER(4). | int2: INTEGER(2) | result: INTEGER(4) |
| **LSTAT** | **LSTAT**(name, statb). Returns information about the file named name in the array statb. | name: CHARACTER*(* | statb(12): INTEGER(4)  
result: INTEGER(4) |
| **LTIME** | **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** | **CALL PERROR**(string). Sends an error message to the standard error stream, preceded by string with a message for the last detected IERRNO error. | string: CHARACTER*(* | |
| **PUTC** | **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 INTEGER(4) argument as an INTEGER(2). | *int4*: INTEGER(4)  
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) |
### Component Object Model (COM) Procedures (DFCOM)

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMAddObjectReference</td>
<td>Function</td>
<td>Adds a reference to an object's interface.</td>
</tr>
<tr>
<td>COMCLSIDFromProgID</td>
<td>Subroutine</td>
<td>Passes a programmatic identifier and returns the corresponding class identifier.</td>
</tr>
<tr>
<td>COMCLSIDFromString</td>
<td>Subroutine</td>
<td>Passes a class identifier string and returns the corresponding class identifier.</td>
</tr>
</tbody>
</table>

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. Programs that use Automation procedures must access the appropriate libraries with USE DFAUTO. Some procedures also require the USE DFWINTY module.
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMCreateObjectByGUID</td>
<td>Subroutine</td>
<td>Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.</td>
</tr>
<tr>
<td>COMCreateObjectByProgID</td>
<td>Subroutine</td>
<td>Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.</td>
</tr>
<tr>
<td>COMGetActiveObjectByGUID</td>
<td>Subroutine</td>
<td>Passes a class identifier and returns a pointer to the interface of a currently active object.</td>
</tr>
<tr>
<td>COMGetActiveObjectByProgID</td>
<td>Subroutine</td>
<td>Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.</td>
</tr>
<tr>
<td>COMGetFileObject</td>
<td>Subroutine</td>
<td>Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.</td>
</tr>
<tr>
<td>COMInitialize</td>
<td>Subroutine</td>
<td>Initializes the COM library.</td>
</tr>
<tr>
<td>COMisEqualGUID</td>
<td>Function</td>
<td>Determines whether two GUIDs are the same.</td>
</tr>
<tr>
<td>COMQueryInterface</td>
<td>Subroutine</td>
<td>Passes an interface identifier and returns a pointer to an object's interface.</td>
</tr>
<tr>
<td>COMReleaseObject</td>
<td>Function</td>
<td>Indicates that the program is done with a reference to an object's interface.</td>
</tr>
<tr>
<td>COMStringFromGUID</td>
<td>Subroutine</td>
<td>Passes a GUID and returns a string of printable characters.</td>
</tr>
<tr>
<td>COMUninitialize</td>
<td>Subroutine</td>
<td>Uninitializes the COM library.</td>
</tr>
</tbody>
</table>

**Automation Server Procedures (DFAUTO)**
<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOAddArg</td>
<td>Subroutine</td>
<td>Passes an argument name and value and adds the argument to the argument list data structure.</td>
</tr>
<tr>
<td>AUTOAllocateInvokeArgs</td>
<td>Function</td>
<td>Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.</td>
</tr>
<tr>
<td>AUTODEallocateInvokeArgs</td>
<td>Subroutine</td>
<td>Deallocates an argument list data structure.</td>
</tr>
<tr>
<td>AUTOGetExceptInfo</td>
<td>Subroutine</td>
<td>Retrieves the exception information when a method has returned an exception status.</td>
</tr>
<tr>
<td>AUTOGetProperty</td>
<td>Function</td>
<td>Passes the name or identifier of the property and gets the value of the Automation object's property.</td>
</tr>
<tr>
<td>AUTOGetPropertyByID</td>
<td>Function</td>
<td>Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.</td>
</tr>
<tr>
<td>AUTOGetPropertyInvokeArgs</td>
<td>Function</td>
<td>Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.</td>
</tr>
<tr>
<td>AUTOInvoke</td>
<td>Function</td>
<td>Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.</td>
</tr>
<tr>
<td>AUTOSetProperty</td>
<td>Function</td>
<td>Passes the name or identifier of the property and a value, and sets the value of the Automation object's property.</td>
</tr>
</tbody>
</table>
For more information on using these procedures, see Using COM and Automation Objects.

**Serial Port Procedures: table**

**Note:** Programs that use `SPORT_nnnn` procedures must access the appropriate libraries with USE DFLIB. The following are all functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT_CONNECT</td>
<td>Establishes the connection to a serial port.</td>
</tr>
<tr>
<td>SPORT_RELEASE</td>
<td>Releases a serial port that has previously been connected.</td>
</tr>
<tr>
<td>SPORT_GET_HANDLE</td>
<td>Returns the WIN32 handle associated with the communications port.</td>
</tr>
<tr>
<td>SPORT_GET_STATE</td>
<td>Returns the baud rate, parity, data bits and stop bit settings of the communications port.</td>
</tr>
<tr>
<td>SPORT_SET_STATE</td>
<td>Sets the baud rate, parity, data bits and stop bit settings of the communications port.</td>
</tr>
<tr>
<td>SPORT_SHOW_STAT</td>
<td>Displays the state of a port.</td>
</tr>
<tr>
<td>SPORT_GET_TIMEOUTS</td>
<td>Returns the user selectable timeouts for the serial port.</td>
</tr>
<tr>
<td>SPORT_SET_TIMEOUTS</td>
<td>Sets the user selectable timeouts for the serial port.</td>
</tr>
<tr>
<td>SPORT_SPECIAL_FUNC</td>
<td>Executes a communications function on a specified port.</td>
</tr>
<tr>
<td>SPORT_WRITE_DATA</td>
<td>Outputs data to a specified port.</td>
</tr>
</tbody>
</table>
For more information on using these procedures, see Using the Serial I/O Port Routines, and Communications and Communications Functions in the "Platform SDK".

### Miscellaneous Run-Time Procedures: table

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FOR_CHECK_FLAWED_PENTIUM</strong></td>
<td>Function</td>
<td><strong>FOR_CHECK_FLAWED_PENTIUM</strong> ( ). Checks the processor to determine if it shows characteristic of the Pentium® floating-point divide flaw.</td>
</tr>
<tr>
<td><strong>FOR_GET_FPE</strong></td>
<td>Function</td>
<td><strong>FOR_GET_FPE</strong> ( ). Returns the current settings of floating-point exception flags.</td>
</tr>
<tr>
<td><strong>FOR_RTL_FINISH</strong></td>
<td>Function</td>
<td><strong>FOR_RTL_FINISH</strong> ( ). 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.</td>
</tr>
</tbody>
</table>
### Functions Not Allowed as Actual Arguments: table

The following specific functions cannot be passed as actual arguments:

<table>
<thead>
<tr>
<th>AIMAX0</th>
<th>EOF</th>
<th>JIDINT</th>
<th>MAX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMIN0</td>
<td>FLOAT</td>
<td>JIFIX</td>
<td>MAX1</td>
</tr>
<tr>
<td>AJMAX0</td>
<td>FLOATI</td>
<td>JINT</td>
<td>MIN0</td>
</tr>
<tr>
<td>AJMIN0</td>
<td>FLOATJ</td>
<td>JMAX0</td>
<td>MIN1</td>
</tr>
<tr>
<td>AKMAX0</td>
<td>FLOATK</td>
<td>JMAX1</td>
<td>MULT_HIGH</td>
</tr>
<tr>
<td>AKMIN0</td>
<td>ICHAR</td>
<td>JMIN0</td>
<td>NUMBER_OF_PROCESSORS</td>
</tr>
<tr>
<td>AMAX0</td>
<td>IDINT</td>
<td>JMIN1</td>
<td>NWORKERS</td>
</tr>
<tr>
<td>AMAX1</td>
<td>IFIX</td>
<td>KIDINT</td>
<td>PROCESSORS SHAPE</td>
</tr>
<tr>
<td>AMIN0</td>
<td>IIDINT</td>
<td>KIFIX</td>
<td>QCMPLX</td>
</tr>
<tr>
<td>AMIN1</td>
<td>IIFIX</td>
<td>KINT</td>
<td>QEXT</td>
</tr>
<tr>
<td>CHAR</td>
<td>IINT</td>
<td>KIQINT</td>
<td>QEXTD</td>
</tr>
<tr>
<td>CMPLX</td>
<td>IMAX0</td>
<td>KIQNNT</td>
<td>QMAX1</td>
</tr>
<tr>
<td>DBLE</td>
<td>IMAX1</td>
<td>KMAX0</td>
<td>QMIN1</td>
</tr>
<tr>
<td>Command</td>
<td>Variable 1</td>
<td>Variable 2</td>
<td>Variable 3</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>DBLEQ</td>
<td>IMIN0</td>
<td>KMAX1</td>
<td>QREAL</td>
</tr>
<tr>
<td>DCMPLX</td>
<td>IMIN1</td>
<td>KMIN0</td>
<td>RAN</td>
</tr>
<tr>
<td>DFLoti</td>
<td>INT</td>
<td>KMIN1</td>
<td>REAL</td>
</tr>
<tr>
<td>DFLotJ</td>
<td>INT_PTR_KIND</td>
<td>LGE</td>
<td>SECNDS</td>
</tr>
<tr>
<td>DFLotK</td>
<td>INT1</td>
<td>LGT</td>
<td>SIZEOF</td>
</tr>
<tr>
<td>DMAX1</td>
<td>INT2</td>
<td>LLE</td>
<td>SNGL</td>
</tr>
<tr>
<td>DMIN1</td>
<td>INT4</td>
<td>LLT</td>
<td>SNGLQ</td>
</tr>
<tr>
<td>DPORD</td>
<td>INT8</td>
<td>LOC</td>
<td>MALLOC</td>
</tr>
<tr>
<td>DREAL</td>
<td>JFIX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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**

```fortran
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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>IIABS</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>IABS ¹</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIABS ²</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>
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:

```fortran
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*(*)\(^\star\). 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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Tests for read permission</td>
</tr>
<tr>
<td>w</td>
<td>Tests for write permission</td>
</tr>
<tr>
<td>x</td>
<td>Tests for execute permission (name must be .COM, .EXE, .BAT, or .CMD)</td>
</tr>
<tr>
<td>(blank)</td>
<td>Tests for existence</td>
</tr>
</tbody>
</table>

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:

- EACCESS: 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, even if the processor's default character set is different. It is the inverse of the **IACHAR** function. In Compaq Fortran, **ACHAR** is equivalent to the **CHAR** 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

\[ \text{result} = \text{ACOS} \ (x) \]

\( x \)

(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 \( \pi \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DACOS</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QACOS (^1)</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

\(^1\) 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

\[ \text{result} = \text{ACOSD} \ (x) \]

\( x \)

(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 \( x \).
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**

\[
    \text{result} = \text{ADJUSTL} (\text{string})
\]

*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`.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG</td>
<td>COMPLEX(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DIMAG</td>
<td>COMPLEX(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QIMAG</td>
<td>COMPLEX(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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:

```fortran
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 of the result is that specified by *kind*; otherwise, the kind parameter is that of *a*.

The result is defined as the largest integer whose magnitude does not exceed the magnitude of *a* and whose sign is the same as that of *a*. If |*a*| is less than 1, AINT (*a*) has the value zero.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINT</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DINT</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QINT ¹</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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**

\[ \text{result} = \text{ALARM} \left( \text{time}, \text{proc} \right) \]

- \(\text{time}\)
  (Input) Integer. Specifies the time delay, in seconds, between the call to \text{ALARM} and the time when \text{proc} is to begin execution. If \text{time} is 0, the alarm is turned off and no routine is called.

- \(\text{proc}\)
  (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 \text{ALARM}), it returns the number of seconds remaining until the previously set alarm is to go off, rounded up to the nearest second.

After \text{ALARM} is called and the timer starts, the calling program continues for \text{time} seconds. The calling program then suspends and calls \text{proc}, which runs in another thread. When \text{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 \text{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 \text{proc} runs in has a higher priority than any other thread in the process. All other threads are essentially suspended until \text{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 \text{EXIT} call, than any pending alarm is deactivated before it has a chance to run.

**Compatibility**
ALARM

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

General Compiler Directive: Declares alternate external names for external subprograms.

Syntax

\texttt{cDEC$ ALIAS internal-name, external-name}

\textit{c}
Is a \texttt{c}, \texttt{C}, !, or *. (See Syntax Rules for General Directives.)

\textit{internal-name}
The name of the subprogram as used in the current program unit.

\textit{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 \textit{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 \texttt{ALIAS} directive affects only the external name used for references to the specified \textit{internal-name}.

Names that are not acceptable to the linker will cause link-time errors.

You can use the prefix \texttt{!MS$} in place of \texttt{cDEC$}.

Compatibility
**ALL**

*Transformational Intrinsic Function (Generic):* Determines if *all* values are true in an entire array or in a specified dimension of an array.

**Syntax**

\[
\text{result} = \text{ALL} (\text{mask} [, \text{dim}] )
\]

- \( \text{mask} \)
  - (Input) Must be a logical array.

- \( \text{dim} \)
  - (Optional; input) Must be a scalar integer with a value in the range 1 to \( n \), where \( n \) is the rank of \( \text{mask} \).

**Results:**

The result is an array or a scalar of type logical.

The result is a scalar if \( \text{dim} \) is omitted or \( \text{mask} \) has rank one. A scalar result is true only if all elements of \( \text{mask} \) are true, or \( \text{mask} \) has size zero. The result has the value false if any element of \( \text{mask} \) is false.

An array result has the same type and kind parameters as \( \text{mask} \), and a rank that is one less than \( \text{mask} \). Its shape is \((d_1, d_2, ..., d_{\text{dim}-1}, d_{\text{dim}+1}, ..., d_n)\), where \((d_1, d_2, ..., d_n)\) is the shape of \( \text{mask} \).

Each element in an array result is true only if all elements in the one dimensional array defined by \( \text{mask} (s_1, s_2, ..., s_{\text{dim}-1}, :, s_{\text{dim}+1}, ..., s_n) \) are true.

**Compatibility**

**Examples**

\[
\text{LOGICAL mask( 2, 3), AR1(3), AR2(2)}
\]
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:

```fortran
REAL, ALLOCATABLE :: Z(:, :, :)
```

The following is an example of the `ALLOCATABLE` statement:

```fortran
REAL A, B(:)
ALLOCATABLE :: A(:,,:), B
```

## ALLOCATE

**Statement:** Dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized.

**Syntax**

```fortran
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:

```fortran
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**

```fortran
!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:

```fortran
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**

```fortran
REAL, ALLOCATABLE :: A(:)

... 
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

Consider the following:

```fortran
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**

```fortran
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**

```fortran
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).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANINT</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DNINT</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QNINT</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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
 amount = 12.99
 taxrate = 0.081
 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

\[
\text{result} = \text{ANY} \left( \text{mask} \ [, \ dim] \right)
\]

\(\text{mask}\)
(Input) Must be a logical array.

\(\text{dim}\)
(Optional; input) Must be a scalar integer expression with a value in the range 1 to \(n\), where \(n\) is the rank of \(\text{mask}\).

Results:

The result is an array or a scalar of type logical.

The result is a scalar if \(\text{dim}\) is omitted or \(\text{mask}\) has rank one. A scalar result is true if any elements of \(\text{mask}\) are true. The result has the value false if no element of \(\text{mask}\) is true, or \(\text{mask}\) has size zero.

An array result has the same type and kind parameters as \(\text{mask}\), and a rank that is one less than \(\text{mask}\). Its shape is \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) is the shape of \(\text{mask}\).

Each element in an array result is true if any elements in the one dimensional array defined by \(\text{mask}\) \((s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n)\) are true.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: ALL, COUNT

Example

\text{LOGICAL mask( 2, 3), AR1(3), AR2(2)}
\text{DATA mask /T, T, F, T, F, F/}
\text{! mask is true false false}
\text{! true false false}
\text{AR1 = ANY(mask, DIM = 1) ! evaluates the elements column by column yielding [true true false]}
\text{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

\[
\begin{bmatrix}
1 & 5 & 7 \\
3 & 6 & 8 
\end{bmatrix}
\]

and B is the array

\[
\begin{bmatrix}
0 & 5 & 7 \\
2 & 6 & 9 
\end{bmatrix}
\]

ANY (A .EQ. B, DIM=1) tests to see if any elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, true) because the second and third columns have at least one element that is equal.

ANY (A .EQ. B, DIM=2) tests to see if any elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (true, true) because each row has at least one element that is equal.

**APPENDMENUQQ**

**QuickWin Function:** Appends a menu item to the end of a menu and registers its callback subroutine.

**Module:** USE DFLIB

**Syntax**

\[
\text{result} = \text{APPENDMENUQQ} (\text{menuID}, \text{flags}, \text{text}, \text{routine})
\]

**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:

- $\text{MENUGRAYED}$ - Disables and grays out the menu item.
- $\text{MENUDISABLED}$ - Disables but does not gray out the menu item.
- $\text{MENUENABLED}$ - Enables the menu item.
- $\text{MENUSEPARATOR}$ - Draws a separator bar.
$\text{MENUCHECKED} - \text{Puts a check by the menu item.} \\
$\text{MENUUNCHECKED} - \text{Removes the check by the menu item.} \\

\text{text} \\
(\text{Input}) \text{ Character*(*)}. \text{ Menu item name. Must be a null-terminated C string, for example, 'WORDS OF TEXT'C.} \\

\text{routine} \\
(\text{Input}) \text{ EXTERNAL}. \text{ 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:} \\

\begin{itemize}
  \item \text{WINPRINT} - \text{Prints the program.} \\
  \item \text{WINSAVE} - \text{Saves the program.} \\
  \item \text{WINEXIT} - \text{Terminates the program.} \\
  \item \text{WINSELECTTEXT} - \text{Selects text from the current window.} \\
  \item \text{WINSELECTGRAPHICS} - \text{Selects graphics from the current window.} \\
  \item \text{WINSELECTALL} - \text{Selects the entire contents of the current window.} \\
  \item \text{WINCOPY} - \text{Copies the selected text and/or graphics from the current window to the Clipboard.} \\
  \item \text{WINPASTE} - \text{Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.} \\
  \item \text{WINCLEARPASTE} - \text{Clears the paste buffer.} \\
  \item \text{WINSIZETOFIT} - \text{Sizes output to fit window.} \\
  \item \text{WINFULLSCREEN} - \text{Displays output in full screen.} \\
  \item \text{WINSTATE} - \text{Toggles between pause and resume states of text output.} \\
  \item \text{WINCASCADE} - \text{Cascades active windows.} \\
  \item \text{WINTILE} - \text{Tiles active windows.} \\
  \item \text{WINARRANGE} - \text{Arranges icons.} \\
  \item \text{WINSTATUS} - \text{Enables a status bar.} \\
  \item \text{WININDEX} - \text{Displays the index for QuickWin help.} \\
  \item \text{WINUSING} - \text{Displays information on how to use Help.} \\
  \item \text{WINABOUT} - \text{Displays information about the current QuickWin application.} \\
  \item \text{NUL} - \text{No callback routine.} \\
\end{itemize}

\text{Results:} \\
The result type is logical. It is .TRUE. if successful; otherwise, .FALSE.. \\

You do not need to specify a menu item number, because \text{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. \text{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** \((x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)\)
result = **ARC_W** \((wx_1, wy_1, wx_2, wy_2, wx_3, wy_3, wx_4, wy_4)\)

\(x_1, y_1\)
(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

\(x_2, y_2\)
(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

\(x_3, y_3\)
(Input) INTEGER(2). Viewport coordinates of start vector.

\(x_4, y_4\)
(Input) INTEGER(2). Viewport coordinates of end vector.

\(wx_1, wy_1\)
(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.

\(wx_2, wy_2\)
(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

\(wx_3, wy_3\)
(Input) REAL(8). Window coordinates of start vector.

\(wx_4, wy_4\)
(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 \((x_1, y_1)\) and \((x_2, y_2)\) for **ARC** and \((wx_1, wy_1)\) and \((wx_2, wy_2)\) for **ARC_W**.

The arc starts where it intersects an imaginary line extending from the center of the arc through \((x_3, y_3)\) for **ARC** and \((wx_3, wy_3)\) 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 \((x_4, y_4)\) for **ARC** and \((wx_4, wy_4)\) 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 below.

```fortran
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**

\[
\text{result} = \text{ASIN} (x)
\]

\(x\)  
(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 \(\pi/2\).
Example

ASIN (0.79345021) has the value 0.9164571.

**ASIND**

Elemental Intrinsic Function (Generic): Produces an arcsine (with the result in degrees).

**Syntax**

\[
\text{result} = \text{ASIND} (x)
\]

\(x\) (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 \(x\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIND</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DASIND</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QASIND</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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**

\[
\text{result} = \text{ASM} \left( \text{string [, a,...]} \right)
\]

*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.
Examples

Consider the following:

! Concatenation is recommended for clarity.
! Notice that ";'" separates instructions.
!
\[
nine=9
\]

\[
type *, asm('addq %0, $17, $0;' //     ! Adds the first two arguments
    'ldq $22, %6;' //         !   and puts the answer in
    'ldq $23, %7;' //         !   register $0
    'ldq $24, %8;' //         !
    'mov $0, %fp;' //         ! Comments are not allowed in the
    'addq %fp, %0;' //         !   constant, but are allowed here
    'addq %0, %fp, %0;' //
    'addq %30, %0, %0;' //
    'addq %31, %0, %0;' //
    'addq %32, %0, %0;' //
    'addq %33, %0, %0;',//
    1,2,3,4,5,6,7,8,nine) // ! The actual arguments to the
    !     ASM (usually by value)
end
\]

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. This feature has been deleted in Fortran 95; it was obsolescent in Fortran 90. Compaq Fortran fully supports features deleted in Fortran 95.

**Syntax**

```fortran
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.

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 was 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:

```fortran
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 \texttt{READ} statement, as in the following example:

\begin{verbatim}
IVBL = 400
\end{verbatim}

The following example shows \texttt{ASSIGN} statements:

\begin{verbatim}
INTEGER ERROR
...
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
ASSIGN 250 TO ERROR
\end{verbatim}

Note that \texttt{NSTART} and \texttt{KSTOP} are integer variables implicitly, but \texttt{ERROR} must be previously declared as an integer variable.

The following statement associates the variable \texttt{NUMBER} with the statement label 100:

\begin{verbatim}
ASSIGN 100 TO NUMBER
\end{verbatim}

If an arithmetic operation is subsequently performed on variable \texttt{NUMBER} (such as follows), the run-time behavior is unpredictable:

\begin{verbatim}
NUMBER = NUMBER + 1
\end{verbatim}

To return \texttt{NUMBER} to the status of an integer variable, you can use the following statement:

\begin{verbatim}
NUMBER = 10
\end{verbatim}

This statement dissociates \texttt{NUMBER} from statement 100 and assigns it an integer value of 10. Once \texttt{NUMBER} is returned to its integer variable status, it can no longer be used in an assigned \texttt{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**

\begin{verbatim}
INTERFACE ASSIGNMENT(=)
\end{verbatim}

The subroutines within the interface block must have two nonoptional arguments, the first with intent \texttt{OUT} or \texttt{INOUT}, and the second with intent \texttt{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:

```fortran
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**:

```fortran
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**:

```fortran
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 SUBROUTINE circle_to_interval

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

\[ \text{variable} = \text{expression} \]

\text{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.

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

```fortran
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
ASSOCIATED

Inquiry Intrinsic Function (Generic): Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.

Syntax

\[
\text{result} = \text{ASSOCIATED (pointer [, target])}
\]

\textit{pointer}

(Input) Must be a pointer (of any data type).

\textit{target}

(Optional; input) Must be a pointer or target. The pointer (in \textit{pointer} or \textit{target}) must not have an association status that is undefined.

Results:

The result type is default logical scalar.

If only \textit{pointer} appears, the result is true if it is currently associated with a target; otherwise, the result is false.

If \textit{target} also appears and is a target, the result is true if \textit{pointer} is currently associated with \textit{target}; otherwise, the result is false.

If \textit{target} is a pointer, the result is true if both \textit{pointer} and \textit{target} are currently associated with the same target; otherwise, the result is false. (If either \textit{pointer} or \textit{target} is disassociated, the result is false.)
The setting of compiler option \texttt{/integer\_size} can affect this function.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} Allocated, Pointer, Target

\textbf{Example}

\begin{verbatim}
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
\end{verbatim}

Consider the following:

\begin{verbatim}
REAL, TARGET, DIMENSION (0:50) :: TAR
REAL, POINTER, DIMENSION (: ) :: PTR
PTR => TAR
PRINT *, ASSOCIATED (PTR, TAR)          ! Returns the value true
\end{verbatim}

The subscript range for PTR is 0:50. Consider the following pointer assignment statements:

\begin{verbatim}
(1) PTR => TAR (:)
(2) PTR => TAR (0:50)
(3) PTR => TAR (0:49)
\end{verbatim}

For statements 1 and 2, \texttt{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, \texttt{ASSOCIATED (PTR, TAR)} is false because the upper bound of TAR has changed.

Consider the following:

\begin{verbatim}
REAL, POINTER, DIMENSION (: ) :: PTR2, PTR3
ALLOCATE (PTR2 (0:15))
PTR3 => PTR2
PRINT *, ASSOCIATED (PTR2, PTR3)          ! Returns the value true
...  
NULLIFY (PTR2)
NULLIFY (PTR3)
PRINT *, ASSOCIATED (PTR2, PTR3)          ! Returns the value false
\end{verbatim}

\textbf{ATAN}
**Elemental Intrinsic Function (Generic):** Produces an arctangent (with the result in radians).

**Syntax**

\[ \text{result} = \text{ATAN} (x) \]

- \( x \) (Input) Must be of type real.

**Results:**

The result type is the same as \( x \). The value lies in the range \(-\pi/2\) to \(\pi/2\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DATAN</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QATAN</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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**

\[ \text{result} = \text{ATAND} (x) \]

- \( x \) (Input) Must be of type real and must be greater than or equal to zero.

**Results:**

The result type is the same as \( x \).
Example

ATAND (0.0874679) has the value 4.998819.

**ATAND**

**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**

\[
\text{result} = \text{ATAND} (y, x)
\]

\(y\)
(Input) Must be of type real.

\(x\)
(Input) Must have the same type and kind parameters as \(y\). If \(y\) has the value zero, \(x\) cannot have the value zero.

**Results:**

The result type is the same as \(x\). The value lies in the range \(-\pi < \text{ATAND} (y, x) \leq \pi\). If \(x \neq 0\), the result is approximately equal to the value of arctan \((y/x)\).

If \(y > 0\), the result is positive.

If \(y < 0\), the result is negative.

If \(y = 0\), the result is zero (if \(x > 0\)) or \(\pi\) (if \(x < 0\)).

If \(x = 0\), the absolute value of the result is \(\pi/2\).
Examples

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y has the value

\[
\begin{bmatrix}
  1 & 1 \\
  -1 & -1
\end{bmatrix}
\]

and X has the value

\[
\begin{bmatrix}
  -1 & 1 \\
  -1 & 1
\end{bmatrix}
\]

then ATAN2 (Y, X) is

\[
\begin{bmatrix}
  \frac{3\pi}{4} & \frac{\pi}{4} \\
  -\frac{3\pi}{4} & -\frac{\pi}{4}
\end{bmatrix}
\]

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**

\[
\text{result} = \text{ATAN2D} \ (x, \ y)
\]

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 \( x \neq 0 \), the result is approximately equal to the value of \( \arctan \left( \frac{y}{x} \right) \).

If \( y > 0 \), the result is positive.

If \( y < 0 \), the result is negative.

If \( y = 0 \), the result is zero (if \( x > 0 \)) or \( 180 \) degrees (if \( x < 0 \)).

If \( x = 0 \), the absolute value of the result is \( 90 \) degrees.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN2D</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DATAN2D</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QATAN2D ¹</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

¹ VMS and U*X

**Examples**

ATAN2D (2.679676, 1.0) has the value 69.53546.

**ATOMIC (TU*X only)**

**OpenMP Parallel Compiler Directive:** Ensures that a specific memory location is updated dynamically; this prevents the possibility of multiple, simultaneous writing threads.

**Syntax**

\[
\text{C}\$\text{OMP ATOMIC}
\]

\[
\text{C}
\]

Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

**Rules and Behavior**

The **ATOMIC** directive permits optimization beyond that of the critical section
around the assignment. An implementation can replace **ATOMIC** directives by enclosing each statement in a critical section. The critical section (or sections) must use the same unique name.

The **ATOMIC** directive applies only to the immediately following statement, which must have one of the following forms:

\[
\begin{align*}
    x &= x \text{ operator } expr \\
    x &= expr \text{ operator } x \\
    x &= \text{ intrinsic } (x, \text{ expr}) \\
    x &= \text{ intrinsic } (\text{expr}, x)
\end{align*}
\]

In the preceding statements:

- *x* is a scalar variable of intrinsic type
- *expr* is a scalar expression that does not reference *x*
- *intrinsic* is **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**
- *operator* is +, *, -, /, .AND., .OR., .EQV., or .NEQV.

All references to storage location *x* must have the same type and type parameters.

Only the loading and storing of *x* are dynamic; the evaluation of *expr* is not dynamic. To avoid race conditions (or concurrency races), all updates of the location in parallel must be protected using the **ATOMIC** directive, except those that are known to be free of race conditions. The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator, and assignment.

**See Also:** Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only)

**Examples**

The following example shows a way to avoid race conditions by using **ATOMIC** to protect all simultaneous updates of the location by multiple threads:

```fortran
    c$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
    DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)
    c$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
    END DO
```

Since the **ATOMIC** directive applies only to the statement immediately following it, note that *Y* is *not* updated atomically.
ATTRIBUTES

General Compiler Directive: Declares properties for specified variables.

Syntax

\[
c \text{DEC$ ATTRIBUTES } att [, att] \ldots :: object [, object] \ldots
\]

\(c\)
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

\(att\)
Is one of the following:

<table>
<thead>
<tr>
<th>Property</th>
<th>ADDRESS64&lt;sup&gt;1&lt;/sup&gt;</th>
<th>DESCRIPTOR32&lt;sup&gt;2&lt;/sup&gt;</th>
<th>REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESCRIPTOR32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIAS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESCRIPTOR64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCE32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAY_VISUALIZER</td>
<td></td>
<td>DLLEXPORT</td>
<td></td>
</tr>
<tr>
<td>DLLIMPORT</td>
<td>DLLIMPORT</td>
<td>STDCALL</td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td>EXTERN</td>
<td>VALUE</td>
</tr>
<tr>
<td>DESCRIPTOR2</td>
<td></td>
<td>NO_ARG_CHECK</td>
<td>VARYING</td>
</tr>
</tbody>
</table>

<sup>1</sup> VMS, WNT
<sup>2</sup> VMS only

\(object\)
Is the name of a data object or procedure.

The following table shows which properties can be used with various objects:

<table>
<thead>
<tr>
<th>Property</th>
<th>Variable and Array Declarations</th>
<th>Common Block Names&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Subprogram Specification and EXTERNAL Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS64</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
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:

```plaintext
MODULE MOD1
  INTERFACE
    SUBROUTINE SUB1
      ALIAS
      ARRAY_VISUALIZER^2
      C
      DEFAULT
      DESCRIPTOR
      DESCRIPTOR32
      DESCRIPTOR64
      DLLEXPORT
      DLLIMPORT
      EXTERN
      NO_ARG_CHECK
      REFERENCE
      REFERENCE32
      REFERENCE64
      STDCALL
      VALUE
      VARYING
  ENDINTERFACE
ENDMOD1
```
In this case, the call to NEW_SUB within SUB2 uses the C and ALIAS properties specified in the interface block.

The properties are described as follows:

- **ADDRESS64 (VMS, WNT)**

  Specifies that the object has a 64-bit address. This property can be specified for any variable or dummy argument, including **ALLOCATABLE** and deferred-shape arrays. However, variables with this property cannot be data-initialized.

  It can also be specified for **COMMON** blocks or for variables in a **COMMON** block. If specified for a COMMON block variable, the COMMON block implicitly has the ADDRESS64 property.

  ADDRESS64 is not compatible with the AUTOMATIC attribute.

- **ALIAS**

  Specifies an alternate external name to be used when referring to external subprograms. Its form is:

  ```
  ALIAS:external-name
  ```

  *external-name*

  Is a character constant delimited by apostrophes or quotation marks. The character constant is used as is; the string is not changed to uppercase, nor are blanks removed.

  The ALIAS property overrides the C (and STDCALL) property. If both C and ALIAS are specified for a subprogram, the subprogram is given the C calling convention, but not the C naming convention. It instead receives the name given for ALIAS, with no modifications.

  ALIAS cannot be used with internal procedures, and it cannot be applied to dummy arguments.

  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

**cDEC$ 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:

  ```fortran
  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 95/90 routines.

  On x86 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:
If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran 95/90 conventions pass arguments by reference.

On x86 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.

<table>
<thead>
<tr>
<th>Convention</th>
<th>C ¹</th>
<th>STDCALL ¹</th>
<th>Default ²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments passed by value</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Case of external subprogram names</td>
<td>VMS: Uppercase</td>
<td>VMS: Uppercase</td>
<td>VMS: Uppercase</td>
</tr>
<tr>
<td></td>
<td>U*X: Lowercase</td>
<td>U*X: Lowercase</td>
<td>U*X: Lowercase</td>
</tr>
<tr>
<td></td>
<td>WNT: Lowercase</td>
<td>WNT: Lowercase</td>
<td>WNT: Lowercase</td>
</tr>
<tr>
<td></td>
<td>W95: Lowercase</td>
<td>W95: Lowercase</td>
<td>W95: Lowercase</td>
</tr>
<tr>
<td>U*X only:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trailing underscore added</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>WNT, W9*:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leading underscore added</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of arguments added</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Caller stack cleanup</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Variable number of arguments</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

¹ C and STDCALL are synonyms on OpenVMS, Tru64 UNIX, and Linux systems, and Windows NT systems on Alpha processors
² The Fortran 95/90 calling convention
Character arguments are passed as follows:

- **By default:**
  - On OpenVMS, Tru64 UNIX, and Linux systems, hidden lengths are put at the end of the argument list.
  - On Windows NT (including Windows 2000) and Windows 9* Systems, hidden lengths immediately follow the variable. To get the OpenVMS, Tru64 UNIX, and Linux behavior, use the `/iface:nomixed_str_len_arg` compiler option.

- **If C or STDCALL (only) is 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 is specified, and REFERENCE is specified for the argument:**
  - On all systems, the string is passed with no length.

- **If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is not specified for the argument, if any):**
  - On all systems, the string is passed with the length.

See also [REFERENCE](#) and [Adjusting Calling Conventions in Mixed-Language Programming](#).

**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 `cDEC$ ATTRIBUTES` options, such as STDCALL, C, REFERENCE, ALIAS, etc. to specify attributes different from the compiler defaults.

This option is useful when declaring **INTERFACE** blocks for external routines, since it prevents compiler options from changing calling or
naming conventions.

- **DESCRIPTOR (VMS only)**

  Specifies that the argument is passed by VMS descriptor. This property can be specified only for dummy arguments in an **INTERFACE** block (*not* for a routine name).

- **DESCRIPTOR32 (VMS only)**

  Specifies that the argument is passed as a 32-bit descriptor.

- **DESCRIPTOR64 (VMS only)**

  Specifies that the argument is passed as a 64-bit descriptor.

- **DLLEXPORT and DLLIMPORT (WNT, W9*)**

  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.

- **NO_ARG_CHECK**
Specifies that type and shape matching rules related to explicit interfaces are to be ignored. This permits the construction of an INTERFACE block for an external procedure or a module procedure that accepts an argument of any type or shape; for example, a memory copying routine.

NO_ARG_CHECK can appear only in an INTERFACE block for a non-generic procedure or in a module procedure. It can be applied to an individual dummy argument name or to the routine name, in which case the property is applied to all dummy arguments in that interface.

NO_ARG_CHECK can not be used for procedures with the PURE or ELEMENTAL prefix. If an argument has an INTENT or OPTIONAL attribute, any NO_ARG_CHECK specification is ignored.

- 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, Tru64 UNIX, and Linux systems, the string is passed with no length.

- On Windows NT (including Windows 2000) and Windows 9* systems, hidden lengths immediately follow the variable. To get the OpenVMS, Tru64 UNIX, and Linux behavior, use the /iface:nomixed_str_len_arg compiler option.
If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine.

If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has not been specified for the argument, the string is passed with the length.

VALUE is the default if the C or STDCALL property is specified in the subprogram definition.

In the following example integer x is passed by value:

```fortran
SUBROUTINE Subr (x)
INTEGER x
!DEC$ ATTRIBUTES VALUE :: x
```

See also C and STDCALL and Adjusting Calling Conventions in Mixed-Language Programming.

- REFERENCE32 (VMS only)
  Specifies that the argument is accepted only by 32-bit address.

- REFERENCE64 (VMS only)
  Specifies that the argument is accepted only by 64-bit address.

- VARYING
  Allows a variable number of calling arguments. If VARYING is specified, the C property must also be specified.

  Either the first argument must be a number indicating how many arguments to process, or the last argument must be a special marker (such as -1) indicating it is the final argument. The sequence of the arguments, and types and kinds must be compatible with the called procedure.

Options C, STDCALL, REFERENCE, VALUE, and VARYING affect the calling conventions of routines. You can specify these cDEC$ ATTRIBUTES options to individual arguments or to an entire routine.

The following form is also allowed:  !MS$ATTRIBUTES att [,att] ... :: object [,object] ...

**Compatibility**
See Also: Programming with Mixed Languages, Creating Fortran DLLs, General Compiler 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 DFWINTY
Syntax

CALL AUTOAddArg (invoke_args, name, value [, intent_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 DFWINTY module.

intent_arg
Indicates the intended use of the argument by the called method. Must be one of the following constants defined in the DFAUTO module:

- AUTO_ARG_IN: The argument's value is read by the called method, but not written. This is the default value if intent_arg is not specified.
- AUTO_ARG_OUT: The argument's value is written by the called method, but not read.
- AUTO_ARG_INOUT: The argument's value is read and written by the called method.

When the value of intent_arg is AUTO_ARG_OUT or AUTO_ARG_INOUT, 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.

type
The variant type of the argument. Must be one of the following constants defined in the DFWINTY module:

<table>
<thead>
<tr>
<th>VARIANT Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_I2</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>VT_I4</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_R4</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>VT_R8</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>
AUTOAllocateInvokeArgs

**DFAUTO Function**: Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.

**Modules**: USE DFAUTO

**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

**Syntax**

CALL AUTODeallocateInvokeArgs(invoke_args)

*invoke_args*

The argument list data structure of type INTEGER(4).

AUTOGetExceptInfo

<table>
<thead>
<tr>
<th>VT_CY</th>
<th>REAL(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_DATE</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>CHARACTER(<em>(</em>))</td>
</tr>
<tr>
<td>VT_DISPATCH</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_ERROR</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>TYPE(VARIANT)</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>INTEGER(4)</td>
</tr>
</tbody>
</table>
**DFAUTO Subroutine**: Retrieves the exception information when a method has returned an exception status.

**Modules**: USE DFAUTO

**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 DFWINTY

**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 following constants defined in the DFWINTY module:

<table>
<thead>
<tr>
<th>VARIANT Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_I2</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>VT_I4</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_R4</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>VT_R8</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>VT_CY</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>VT_DATE</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>CHARACTER*(*)</td>
</tr>
<tr>
<td>VT_DISPATCH</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_ERROR</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>INTEGER(4)</td>
</tr>
</tbody>
</table>

**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

**Syntax**

```plaintext
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

**Syntax**

```plaintext
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

**Syntax**

\[
\text{result} = \text{AUTOInvoke}(\text{idispatch, id, invoke_args})
\]

\text{idispatch}  
The object's IDispatch interface pointer. Must be of type INTEGER(4).

\text{id}  
The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

\text{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:**

\[
type, [\text{att-ls},] \text{AUTOMATIC} [\text{att-ls},] :: v [, v] ...
\]
**Statement:**

\[ \text{AUTOMATIC} [::] \; v \; [, \; v] \; ... \]

*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...
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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>AUTOMATIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy argument</td>
<td>No</td>
</tr>
<tr>
<td>Automatic object</td>
<td>No</td>
</tr>
<tr>
<td>Common block item</td>
<td>No</td>
</tr>
<tr>
<td>Use-associated item</td>
<td>No</td>
</tr>
<tr>
<td>Function result</td>
<td>No</td>
</tr>
<tr>
<td>Component of a derived type</td>
<td>No</td>
</tr>
</tbody>
</table>

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:

```fortran
REAL, AUTOMATIC :: A, B, C
```

The following example uses an AUTOMATIC statement:

```fortran
... CONTAINS
   INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
```
In this example, all variables within the program unit are automatic, except for "var1" and "var2"; these are explicitly declared in a SAVE statement, and thus have static memory locations:

```fortran
SUBROUTINE DoIt (arg1, arg2)
  INTEGER(4) arg1, arg2
  INTEGER(4) var1, var2, var3, var4

  AUTOMATIC
  SAVE var1, var3

  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 DFWINTY

**Syntax**

```fortran
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 following constants defined in the DFWINTY module:

<table>
<thead>
<tr>
<th>VARIANT Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_I2</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>VT_I4</td>
<td>INTEGER(4)</td>
</tr>
</tbody>
</table>
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

**Syntax**

\[
\text{result} = \text{AUTOSetPropertyByID}(\text{idispacht}, \text{memid}, \text{invoke\_args})
\]

\text{idispacht}
The object's IDispatch interface pointer. Must be of type INTEGER(4).

\text{memid}
Member ID of the property. Must be of type INTEGER(4).

\text{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

**Syntax**

\[
\text{result} = \text{AUTOSetPropertyInvokeArgs} \left( \text{idispatch}, \text{invoke\_args} \right)
\]

- \text{idispatch}
The object's IDispatch interface pointer. Must be of type INTEGER(4).

- \text{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**

\[
\text{BACKSPACE} \left( \left[ \text{UNIT=} \right] \text{io\_unit} \left[ , \text{ERR=} \text{label} \right] \left[ , \text{IOSTAT=} \text{i\_var} \right] \right)
\]

- \text{io\_unit}
  (Input) Is an external unit specifier.

- \text{label}
  Is the label of the branch target statement that receives control if an error occurs.

- \text{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 READs 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.
**BARRIER** *(TU*X only)*

**OpenMP Parallel Compiler Directive:** Synchronizes all the threads in a team. It causes each thread to wait until all of the other threads in the team have reached the barrier.

**Syntax**

```c
$OMP BARRIER
```

- `C` is one of the following: `C` (or `c`), `!`, or `*` (see Syntax Rules for Parallel Directives).

The **BARRIER** directive must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

**See Also:** Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives *(TU*X only)*, Compaq Fortran Parallel Compiler Directives *(TU*X only)*, Nesting and Binding Rules

**Examples**

The directive binding rules call for a **BARRIER** directive to bind to the closest enclosing **PARALLEL** directive. In the following example, the **BARRIER** directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

```c
$OMP PARALLEL
$OMP DO PRIVATE(i)
  DO i = 1, 100
    b(i) = i
  END DO
$OMP BARRIER
$OMP DO PRIVATE(i)
  DO i = 1, 100
    a(i) = b(101-i)
  END DO
$OMP END PARALLEL
```

**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

\[
\text{result} = \text{BIT}(\text{bitnum}, \text{source})
\]

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 \text{bitnum} of \text{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

\[
\text{result} = \text{BIT\_SIZE}(i)
\]

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:

<table>
<thead>
<tr>
<th>COMMON</th>
<th>INTRINSIC</th>
<th>STATIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>PARAMETER</td>
<td>TARGET</td>
</tr>
<tr>
<td>Derived-type definition</td>
<td>POINTER</td>
<td>Type declaration ²</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>RECORD ¹</td>
<td>USE ³</td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>Record structure declaration ¹</td>
<td></td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>SAVE</td>
<td></td>
</tr>
</tbody>
</table>

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
...
The following block-data subprogram initializes the named common block /Lakes/:

```plaintext
COMMON /Lakes/ erie, huron, michigan, ontario, superior, fname
DATA erie, huron, michigan, ontario, superior /1, 2, 3, 4, 5/
CHARACTER(LEN=10) fname/'GreatLakes'/
```

---

**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**

```plaintext
result = BSEARCHQQ (adrkey, adrray, length, size)
```

- `adrkey` (Input) INTEGER(4). Address of the variable containing the element to be found (returned by LOC).
- `adrray` (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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type of array</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRT$INTEGER1</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>SRT$INTEGER2</td>
<td>INTEGER(2) or equivalent</td>
</tr>
<tr>
<td>SRT$INTEGER4</td>
<td>INTEGER(4) or equivalent</td>
</tr>
<tr>
<td>SRT$REAL4</td>
<td>REAL(4) or equivalent</td>
</tr>
<tr>
<td>SRT$REAL8</td>
<td>REAL(8) or equivalent</td>
</tr>
</tbody>
</table>
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**

\[ \text{result} = \text{BTEST} \left( i, pos \right) \]
\( i \)
(Input) Must be of type integer.

\( pos \)
(Input) Must be of type integer. It must not be negative and it must be less than \texttt{BIT\_SIZE}(\( i \)).
The rightmost (least significant) bit of \( i \) is in position 0.

\textbf{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 more information, see \texttt{Bit Functions}.

For information on the model for the interpretation of an integer value as a sequence of bits, see \texttt{Model for Bit Data}.

The setting of compiler option \texttt{/integer\_size} can affect this function.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>BITEST</td>
<td>INTEGER(2)</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>BTEST</td>
<td>INTEGER(4)</td>
<td>LOGICAL(4)</td>
</tr>
<tr>
<td>BKTEST</td>
<td>INTEGER(8)</td>
<td>LOGICAL(8)</td>
</tr>
</tbody>
</table>

\textsuperscript{1} Or BJTEST
\textsuperscript{2} Alpha only

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{IBCLR}, \texttt{IBSET}, \texttt{IBCHNG}, \texttt{IOR}, \texttt{IEOR}, \texttt{IAND}

\textbf{Examples}

BTEST (9, 3) has the value true.

If A has the value

\[
\begin{bmatrix}
  1 & 2 
\end{bmatrix}
\]
the value of BTEST (A, 2) is

\[
\begin{bmatrix}
false & false \\
false & true
\end{bmatrix}
\]

and the value of BTEST (2, A) is

\[
\begin{bmatrix}
true & false \\
false & false
\end{bmatrix}
\]

The following shows more examples:

<table>
<thead>
<tr>
<th>Function reference</th>
<th>i</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST (i,2)</td>
<td>00011100 0111000</td>
<td>\texttt{.FALSE.}</td>
</tr>
<tr>
<td>BTEST (i,3)</td>
<td>00011100 0111000</td>
<td>\texttt{.TRUE.}</td>
</tr>
</tbody>
</table>

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 or other external procedure, or a dummy argument associated with a subroutine subprogram or other external procedure.

a-arg
Is an actual argument optionally preceded by [keyword=], where keyword is the name of a dummy argument in the explicit interface for the subroutine. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, the name of a procedure, or an alternate return specifier. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

An alternate return specifier is an asterisk (*), or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90.)

Rules and Behavior

When the CALL statement is executed, any expressions in the actual argument list are evaluated, then control is passed to the first executable statement or construct in the subroutine. When the subroutine finishes executing, control returns to the next executable statement following the CALL statement, or to a statement identified by an alternate return label (if any).

If an argument list appears, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.
An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see 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 alternate returns
CALL SUBA(X, &30, &50, Y)

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)
END PROGRAM KEYWORD_EXAMPLE

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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>A Match Occurs If:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>case-index .EQV. case-value</td>
</tr>
<tr>
<td>Integer or Character</td>
<td>case-index = = case-value</td>
</tr>
</tbody>
</table>

- When the case value is a range of values (a colon appears), a match depends on the range specified, as follows:
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**

<table>
<thead>
<tr>
<th>Construct</th>
<th>Flow of Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT CASE (TEST 1)</td>
<td></td>
</tr>
<tr>
<td>CASE (1) block 1</td>
<td></td>
</tr>
<tr>
<td>CASE (2) block 2</td>
<td></td>
</tr>
<tr>
<td>END SELECT</td>
<td></td>
</tr>
</tbody>
</table>

Evaluate Test 1

- **Matches** CASE (1)
  - Yes: Execute block 1
  - No: Matches CASE (2)
    - Yes: Execute block 2
    - No: EXECUTE
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:

```plaintext
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
```
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**

\[
\text{result} = \text{CEILING} \left( a \ [, \ kind]\right)
\]

- \(a\) (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 \textit{kind} is present, the kind parameter is that specified by \textit{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:

\begin{verbatim}
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)
\end{verbatim}

\textbf{CHANGEDIRQQ}

\textbf{Run-Time Function:} Makes the specified directory the current, default directory.

\textbf{Module:} USE DFLIB

\textbf{Syntax}

\begin{verbatim}
result = CHANGEDIRQQ (dir)
\end{verbatim}

\textit{dir}

(Input) Character*(*). Directory to be made the current directory.

\textbf{Results}

\texttt{LOGICAL}(4). \texttt{.TRUE.} if successful; otherwise, \texttt{.FALSE.}

If you do not specify a drive in the \textit{dir} string, the named directory on the current drive becomes the current directory. If you specify a drive in \textit{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 = CHANGEDRI
```

CHAR

**Elemental Intrinsic Function (Generic):** Returns the character in the specified position of the processor's 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 processor's 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 processor's character set. **ICHAR** (**CHAR(i, kind(c))**) has the value 1 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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2)</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>CHAR</td>
<td>INTEGER(4)</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>INTEGER(8)</td>
<td>CHARACTER</td>
</tr>
</tbody>
</table>

1 This specific function cannot be passed as an actual argument.  
2 INTEGER(8) is only available on Alpha processors.

**Compatibility**
CHAR

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

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
C   CHARACTER wt*10, city*80, ch
C   CHARACTER (LEN = 10), PRIVATE :: vs
C   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

Length of susan, patty, and dotty are 2, alice is 12,
jane is a 79-member array of length 2

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: '

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**

\[
\text{result} = \text{CHMOD} \left( \text{name, mode} \right)
\]

**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:

<table>
<thead>
<tr>
<th>Permission bit</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>Set user ID on execution</td>
<td>ignored; never true</td>
</tr>
<tr>
<td>2000</td>
<td>Set group ID on execution</td>
<td>ignored; never true</td>
</tr>
<tr>
<td>1000</td>
<td>Sticky bit</td>
<td>ignored; never true</td>
</tr>
<tr>
<td>0400</td>
<td>Read by owner</td>
<td>ignored; always true</td>
</tr>
<tr>
<td>0200</td>
<td>Write by owner</td>
<td>Settable</td>
</tr>
<tr>
<td>0100</td>
<td>Execute by owner</td>
<td>ignored; based on filename</td>
</tr>
<tr>
<td>0040, 0020, 0010</td>
<td>Read, Write, Execute by group</td>
<td>ignored; assumes owner permissions</td>
</tr>
<tr>
<td>0004, 0002, 0001</td>
<td>Read, Write, Execute by others</td>
<td>ignored; assumes owner permissions</td>
</tr>
</tbody>
</table>

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

**CHUNK (TU*X only)**

**Compaq Fortran Parallel Compiler Directive:** Sets a default chunksize to adjust the number of iterations assigned to a thread. The effect of CHUNK varies, depending on the scheduling type.

**Syntax**

\[ \texttt{c$PAR CHUNK = chunksize} \]

C
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).
chunksize

Is a scalar integer expression.

**Rules and Behavior**

The effect of `chunksize` varies by scheduling type, as follows:

- For DYNAMIC and INTERLEAVED scheduling, iterations are always assigned to threads in `chunksize` groups. If the total number of iterations is not divisible by `chunksize`, the last group has fewer than `chunksize` iterations.

- For GUIDED scheduling, `chunksize` is the minimum number of iterations that can be assigned to a thread. If less than `chunksize` iterations remain, the next available thread is assigned all of the remaining iterations.

- For STATIC scheduling, `chunksize` is ignored and has no effect.

The `chunksize` used for any parallel DO loop is determined by the following (in the order shown):

1. A `chunksize` specified in the PDO directive for the current DO loop
2. A user-specified default specified in the most recent CHUNK directive
3. If the scheduling for the current DO loop is INTERLEAVED, DYNAMIC, GUIDED, or RUNTIME, a user-specified default specified in the environment variable MP_CHUNK
4. The compiler default of one

The following form is also allowed: `c$CHUNK = chunksize`

**See Also:** MP_SCHEDTYPE, Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

**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)

CLEARSTATUSFPQQ (x86 only)

Run-Time Subroutine: Clears the exception flags in the floating-point processor status word. This routine is only available on x86 processors.

Module: USE DFLIB

Syntax

CALL CLEARSTATUSFPQQ ( )

The floating-point status word indicates which floating-point exception conditions have occurred. Visual Fortran initially clears (sets to 0) all floating-point status flags, but as exceptions occur, the status flags accumulate until the program clears the flags again. CLEARSTATUSFPQQ will clear the flags.

CLEARSTATUSFPQQ is appropriate for use in applications that poll the floating-point status register as the method for detecting a floating-point exception has occurred.
For a full description 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: GETSTATUSFPQQ, SETCONTROLFPQQ, GETCONTROLFPQQ, SIGNALQQ, MATHERRQQ

Example

! Program to demonstrate CLEARSTATUSFPQQ.
! This program uses polling to detect that a floating-point exception has occurred.
! So, build this console application with the default floating-point exception behavior, fpe3.
PROGRAM CLEARFP
USE DFLIB
REAL*4 A,B,C
INTEGER*2 STS

A = 2.0E0
B = 0.0E0

! Poll and display initial floating point status
CALL GETSTATUSFPQQ(STS)
WRITE(*,'(1X,A,Z4.4)') 'Initial fp status = ',STS

! Cause a divide-by-zero exception
! Poll and display the new floating point status
C = A/B
CALL GETSTATUSFPQQ(STS)
WRITE(*,'(1X,A,Z4.4)') 'After div-by-zero fp status = ',STS

! If a divide by zero error occurred, clear the floating point status register so future exceptions can be detected.
IF ((STS .AND. FPSW$ZERODIVIDE) > 0) THEN
    CALL CLEARSTATUSFPQQ()
    CALL GETSTATUSFPQQ(STS)
    WRITE(*,'(1X,A,Z4.4)') 'After CLEARSTATUSFPQQ fp status = ',STS
ENDIF
END

This program is available in the online samples.

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**

\[
\text{result} = \text{CLICKMENUQQ}(\text{item})
\]

*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**

\[
\text{result} = \text{CLOCK}()
\]

**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**

```fortran
USE DFPORT
character(8) whatimeisit
whatimeisit = CLOCK ()
write *, 'The current time is ',whatimeisit
```

**CLOSE**

**Statement:** Disconnects a file from a unit.

**Syntax**

```fortran
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.
Is the label of the branch target statement that receives control if an error occurs.

(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
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. This function must not be passed as an actual argument.

**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 the result value and zero is assigned to the imaginary part. If y is not specified and x is complex, the result value is `CMPLX (REAL(x), AIMAG(x))`.

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

`CMPLX(x, y, kind)` has the complex value whose real part is `REAL(x, kind)` and whose imaginary part is `REAL(y, kind)`.

The setting of compiler 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:

```plaintext
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

**Syntax**

```plaintext
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.

**See Also:**

[IUnknown::AddRef](https://docs.microsoft.com/en-us/windows/win32/api/unkbase/nf-unkbase-iunknown-addref) in the Win32 SDK

COMCLSIDFromProgID

**DFCOM Subroutine:** Passes a programmatic identifier and returns the corresponding class identifier.

**Modules:** USE DFCOM, USE DFWINTY

**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 DFWINTY module.

status
The status of the operation. It can be any status returned by CLSIDFromProgID. Must be of type INTEGER(4).

See Also:

CLSIDFromProgID in the Win32 SDK

COMCLSIDFromString

DFCOM Subroutine: Passes a class identifier string and returns the corresponding class identifier.

Modules: USE DFCOM, USE DFWINTY

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 DFWINTY module.

status
The status of the operation. It can be any status returned by CLSIDFromString. Must be of type INTEGER(4).

See Also:

CLSIDFromString in 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 DFWINTY

**Syntax**

```call
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 DFWINTY 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 DFWINTY module.

- **iid**
  The interface identifier of the interface being requested. Must be of type GUID, which is defined in the DFWINTY 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).

**See Also**:

CoCreateInstance in 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

**Syntax**

```call
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).

See Also:

**CLSIDFromProgID** and **CoCreateInstance** in the Win32 SDK

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 DFWINTY

**Syntax**

```call CALL COMGetActiveObjectByGUID (clsid, 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 DFWINTY module.

**iid**
The interface identifier of the interface being requested. Must be of type GUID, which is defined in the DFWINTY 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 **GetActiveObject**. Must be of type INTEGER(4).
**COMGetActiveObjectByGUID**

**See Also:**

SetActiveObject in 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

**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 GetActiveObject. Must be of type INTEGER(4).

**See Also:**

CLSIDFromProgID and GetActiveObject in 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

**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 the CreateBindCtx or MkParseDisplayName routines, or the IMoniker::BindToObject method. Must be of type INTEGER(4).

See Also:
CreateBindCtx, MkParseDisplayName, and IMoniker::BindToObject in the Win32 SDK

COMInitialize

DFCOM Subroutine: Initializes the COM library.

Modules: USE DFCOM

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.

See Also:
OleInitialize in the Win32 SDK

COMIsEqualGUID

DFCOM Function: Determines whether two globally unique identifiers (GUIDs) are the same.

Modules: USE DFCOM, USE DFWINTY

Syntax

CALL COMIsEqualGUID (guid1, guid2)

guid1
The first GUID. Must be of type GUID, which is defined in the DFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).

\textit{guid2}

The second GUID, which will be compared to \textit{guid1}. It must be the same type of GUID as \textit{guid1}. For example, if \textit{guid1} is a CLSID, \textit{guid2} must also be a CLSID.

**Results:**

The result type is LOGICAL(4). The result is .TRUE. if the two GUIDs are the same; otherwise, .FALSE.

**See Also:**

\texttt{IsEqualGUID} in 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**

\[
\text{result} = \text{COMMITQQ} (\text{unit})
\]

\textit{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 initially 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. \texttt{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.

**COMMIT** 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='COMMIT.TST', ACCESS='Sequential')
DO WHILE (.TRUE.)
  WRITE (*, '(A, )') 'Enter some data (Hit RETURN to &
  exit): '
  len = GETSTR (stuff)
  IF (len .EQ. 0) EXIT
  WRITE (unit, *) stuff
  IF (.NOT. COMMIT(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, and must not have the same name as any other global entity in the program, such as a subroutine or function.

Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name. 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:
So, variables can be associated if they are of different numeric type. For example, the following is valid:

```fortran
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

```fortran
COMMON CENTS
```

### Program Unit B

```fortran
INTEGER(2) MONEY
COMMON /QUANTA/ A, Y
```

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`. 

---

**Table: Type of Variable and Associated Variable**

<table>
<thead>
<tr>
<th>Type of Variable</th>
<th>Type of Associated Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrinsic numeric 1 or numeric sequence 2</td>
<td>Can be of any of these types</td>
</tr>
<tr>
<td>Default character or character sequence 2</td>
<td>Can be of either of these types</td>
</tr>
<tr>
<td>Any other intrinsic type</td>
<td>Must have the same type and kind parameters</td>
</tr>
<tr>
<td>Any other sequence type</td>
<td>Must have the same type</td>
</tr>
</tbody>
</table>

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.
Named common blocks must be declared to have the same size in each program unit. Blank common can have different lengths in different program units.

**Note:** On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* 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**

```fortran
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

```fortran
COMMON HEAT, X /BLK1/KILO, Q
...
```

### Subprogram

```fortran
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:

```fortran
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:

```fortran
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**

```fortran
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**

```fortran
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 DFWINTY

**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 DFWINTY 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).
See Also:

[IUnknown::QueryInterface](#) in the Win32 SDK

**COMReleaseObject**

**DFCOM Function**: Indicates that the program is done with a reference to an object's interface.

**Modules**: USE DFCOM

**Syntax**

```plaintext
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.

**COMStringFromGUID**

**DFCOM Subroutine**: Passes a globally unique identifier (GUID) and returns a string of printable characters.

**Modules**: USE DFCOM, USE DFWINTY

**Syntax**

```plaintext
CALL COMStringFromGUID (guid, string, status)
```

*guid*

The GUID to be converted. Must be of type GUID, which is defined in the DFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).

*string*

A character variable of type CHARACTER*(*)(*) that receives the string representation of the GUID. The length of the character variable should be at least 38.

*status*

The status of the operation. If the string is too small to contain the string representation of the GUID, the value is zero. Otherwise, the value is the number of characters in the string representation of the GUID. Must be of
type INTEGER(4).

The string representation of a GUID has a format like that of the following:

[c200e360-38c5-11ce-ae62-08002b2b79ef]

where the successive fields break the GUID into the form DWORD-WORD-WORD-WORD-WORD-DWORD covering the 128-bit GUID. The string includes enclosing braces, which are an OLE convention.

See Also:

[StringFromGUID2](#) in the Win32 SDK

**COMUninitialize**

**DFCOM Subroutine**: Uninitializes the COM library.

**Modules**: USE DFCOM

**Syntax**

```call
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**

```call
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 \((x, -y)\).
### 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:

```plaintext
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**

```plaintext
CONTAINS
```

Any number of internal procedures can follow a CONTAINS statement, but a CONTAINS statement cannot appear in the internal procedures themselves.

**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 array(10)
    DO 100 n = 1, 10
    array(n) = 120
100 CONTINUE
COPYIN (TU*X only)

Compaq Fortran Parallel Compiler Directive: Copies the values of listed data objects from the master thread to PRIVATE data objects of the same name in slave threads.

Syntax

\[
c$\text{PAR COPYIN object[, object]} \ldots
\]

\(c\)
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\(\text{object}\)
Is the name of a variable, a single array element, or a named common block. A named common block must appear between slashes (//).

Rules and Behavior

Single array elements can be copied, but array sections cannot be copied.

\text{SHARE​D} variables cannot be copied.

When an \text{ALLOCATABLE} array is to be copied, it must be allocated when the COPYIN directive is encountered.

COPYIN directives are permitted only within \text{PARALLEL} or \text{PARALLEL DO} constructs.

The following form is also allowed: \(c$\text{COPYIN object[, object]} \ldots\)

See Also: Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

Examples

Consider the following:

\[
\text{CPAR$ COPYIN A, B, /X/, C(I)}
\]

This statement specifies that the following be copied from the master thread to the PRIVATE data objects of the same name: A and B, the entire contents of common block X, and the \text{I}th element of C.
**COPYIN Clause (TU*X only)**

**Parallel Directive Clause:** Specifies that the data in the master thread of the team is to be copied to the thread private copies of the common block at the beginning of the parallel region.

**Syntax**

```
COPYIN (list)
```

`list`  
Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The `COPYIN` clause applies only to common blocks declared as THREADPRIVATE (or TASKCOMMON).

You do not need to specify the whole THREADPRIVATE (or TASKCOMMON) common block, you can specify named variables within the common block.

**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`. 

### 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**

\[
\text{result} = \text{COSD} (x)
\]

\(x\)

(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 \(x\).
COSD

**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**

\[ \text{result} = \text{COSH} (x) \]

\( x \)

(Input) Must be of type real.

**Results:**

The result type is the same as \( x \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSH</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DCOSH</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QCOSH ¹</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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**

\[
\text{result} = \text{COTAN} (x)
\]

\[x\] (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 \(x\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTAN</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DCOTAN</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QCOTAN (^1)</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

\(^1\) 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**

\[
\text{result} = \text{COTAND} (x)
\]
\( x \)

(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 \( x \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTAND</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DCOTAND</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QCOTAND ¹</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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**

\[
\text{result} = \text{COUNT} (\text{mask} [, \text{dim}])
\]

\( \text{mask} \)

(Input) Must be a logical array.

\( \text{dim} \)

(Optional; input) Must be a scalar integer expression with a value in the range 1 to \( n \), where \( n \) is the rank of \( \text{mask} \).

**Results:**

The result is an array or a scalar of type default integer.

The result is a scalar if \( \text{dim} \) is omitted or \( \text{mask} \) has rank one. A scalar result has
a value equal to the number of true elements of \textit{mask}. If \textit{mask} has size zero, the result is zero.

An array result has a rank that is one less than \textit{mask}, and shape \((d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) is the shape of \textit{mask}.

Each element in an array result equals the number of elements that are true in the one dimensional array defined by \textit{mask} \((s_1, s_2, \ldots, s_{dim-1}, :, s_{dim+1}, \ldots, s_n)\).

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} ALL, ANY

\textbf{Example}

COUNT \((\text{/.TRUE., .FALSE., .TRUE./})\) has the value 2 because two elements are true.

COUNT \((\text{/.TRUE., .TRUE., .TRUE./})\) has the value 3 because three elements are true.

A is the array

\begin{align*}
\begin{bmatrix}
1 & 5 & 7 \\
3 & 6 & 8 \\
\end{bmatrix}
\end{align*}

and B is the array

\begin{align*}
\begin{bmatrix}
0 & 5 & 7 \\
2 & 6 & 9 \\
\end{bmatrix}
\end{align*}

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 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 new intrinsic subroutine in Fortran 95.

**Syntax**

```fortran
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:

```fortran
REAL time_begin, time_end
...
CALL CPU_TIME ( time_begin )
!
!task to be timed
!
CALL CPU_TIME ( time_end )
PRINT *, 'Time of operation was ', time_end - time_begin, ' seconds'
```

**CRITICAL (TU*X only)**

**OpenMP Parallel Compiler Directive:** Restricts access to a block of code to
only one thread at a time.

Syntax

\[
\text{c}\$\text{OMP CRITICAL} \ [(\text{name})] \\
\text{block} \\
\text{c}\$\text{OMP END CRITICAL} \ [(\text{name})]
\]

c
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

name
Is the name of the critical section.

block
Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name. All unnamed CRITICAL directives map to the same name.

If a name is specified in the CRITICAL directive, the same name must appear in the corresponding END CRITICAL directive. If no name appears in the CRITICAL directive, no name can appear in the corresponding END CRITICAL directive.

Critical section names are global entities of the program. If the name specified conflicts with any other entity, the behavior of the program is undefined.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only)

Examples

The following example shows a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation is placed in a critical section.

Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, XAXIS and YAXIS, respectively:

\[
\text{c}\$\text{OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)}
\]
CRITICAL SECTION (TU*X only)

Compaq Fortran Parallel Compiler Directive: Restricts access to a block of code to only one thread at a time.

Syntax

\[
c$\text{PAR CRITICAL SECTION} \ [(latch-var)]
block
\]

\[
c$\text{PAR END CRITICAL SECTION}
\]

\[
\]

\[
c
\]

Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\[
latch-var
\]

Is a naturally aligned INTEGER(4) or INTEGER(8) SHARED variable.

\[
block
\]

Is one or more statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

Critical sections can appear anywhere an executable Compaq Fortran statement can appear.

You can use a latch variable to control whether multiple critical sections have unique latches or use the same latch. If a latch variable name is used, the program must explicitly initialize the latch variable to zero before any CRITICAL SECTION using that latch variable is executed.

If you do not specify a latch variable, the compiler supplies a unique latch for each critical section.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same latch variable name. When the thread that is executing the critical section reaches the END CRITICAL SECTION directive, the latch variable is set to 1, allowing another thread to
enter the critical section.

The program must not reuse that latch variable in anything other than a CRITICAL SECTION until all uses as a latch variable are complete.

All unnamed critical sections map to the same latch variable name supplied by the compiler. Critical section latch variable names are global to the program.

**See Also:** Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

**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**

\[
\text{result} = \text{CSHIFT} (\text{array}, \text{shift} [, , \text{dim}])
\]

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 \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) 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 \((1 + \text{MODULO} (i + \text{shift} - 1, \text{SIZE (array)}))\). (The same shift is applied to each element.)
If array has rank greater than one, each section \((s_1, s_2, \ldots, s_{dim-1}, :\), \(s_{dim+1}, \ldots, s_n)\) of the result is shifted as follows:

- By the value of shift, if shift is scalar
- According to the corresponding value in shift\((s_1, s_2, \ldots, s_{dim-1}, s_{dim+1}, \ldots, s_n)\), 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 \((V, \text{SHIFT}=2)\) 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 \((V, \text{SHIFT}= -2)\) 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

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

CSHIFT \((M, \text{SHIFT} = 1, \text{DIM} = 2)\) produces the result

\[
\begin{bmatrix}
2 & 3 & 1 \\
5 & 6 & 4 \\
8 & 9 & 7 \\
\end{bmatrix}
\]

Each element in rows 1, 2, and 3 is shifted to the left by 2 positions. The elements shifted off the beginning are inserted at the end.
CSHIFT (M, SHIFT = -1, DIM = 1) produces the result

\[
\begin{bmatrix}
7 & 8 & 9 \\
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{bmatrix}
\]

Each element in columns 1, 2, and 3 is shifted down by 1 position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, -1, 0/), DIM = 2) produces the result

\[
\begin{bmatrix}
2 & 3 & 1 \\
6 & 4 & 5 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

Each element in row 1 is shifted to the left by 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**

\[
\text{result} = \text{CTIME} \left( \text{stime} \right)
\]

\text{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 

\texttt{stime}

\ can be determined by calling the \texttt{TIME} function. \texttt{CTIME (TIME( ))} returns the current time and date.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \texttt{DATE\_AND\_TIME}

Example

\begin{verbatim}
character (24) systime
systime = CTIME (TIME( ))
print *, 'Current date and time is ', systime
\end{verbatim}

\section*{CYCLE}

Statement: Interrupts the current execution cycle of the innermost (or named) \texttt{DO} construct.

Syntax

\begin{verbatim}
CYCLE [ name ]
\end{verbatim}

\begin{itemize}
  \item \texttt{name} (Optional) Is the name of the \texttt{DO} construct.
\end{itemize}

Rules and Behavior

When a \texttt{CYCLE} statement is executed, the following occurs:

\begin{enumerate}
  \item The current execution cycle of the named (or innermost) \texttt{DO} construct is terminated.

  \begin{itemize}
    \item If a \texttt{DO} construct name is specified, the \texttt{CYCLE} statement must be within the range of that construct.
  \end{itemize}

  \item The iteration count (if any) is decremented by 1.
\end{enumerate}
3. The DO variable (if any) is incremented by the value of the increment parameter (if any).


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:

```fortran
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:

```fortran
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 variables or implied-do lists, separated by commas.

Subscript expressions and expressions in substring references must be initialization expressions.

An implied-do list in a **DATA** statement takes the following form:

\[(do\text{-}list, \text{var} = \text{expr}_1, \text{expr}_2 [, \text{expr}_3])\]

\*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.

**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:

```plaintext
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:
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:

```
INTGGER A(10), B(10)
CHARACTER BELL, TAB, LF, FF, STARS*6
DATA A,STARS /10*0,'****'/
```

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 /
```

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
```

**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

```plaintext
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**

```plaintext
CALL DATE (string)
```

**Function Syntax**

```plaintext
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 March 02, 2000, the following
!code prints "02-Mar-00"
CHARACTER(9) TODAY
CALL DATE(TODAY)
PRINT *, TODAY
!The next line prints "03/02/00"
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 2000 March 28 at 11:04:14.5:

```fortran
INTEGER DATE_TIME (8)
CHARACTER (LEN = 12) REAL_CLOCK (3)
CALL DATE_AND_TIME (REAL_CLOCK (1), REAL_CLOCK (2), &
                    REAL_CLOCK (3), DATE_TIME)
```

This assigns the value "20000328" to *REAL_CLOCK (1)*, the value "110414.500" to *REAL_CLOCK (2)*, and the value ":-0500" to *REAL_CLOCK (3)*. The following values are assigned to *DATE_TIME*: 2000, 3, 28, -300, 11, 4, 14, and 500.

The following shows another example:

```fortran
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

\[
\text{result} = \text{DBLE}(a)
\]

\(a\)

(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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(4)</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>
DBLE

### 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. This function must not be specified as an actual argument.

**Syntax**

\[
\text{result} = \text{DCMPLX} \ (x \ [, \ y])
\]

\(x\)
(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 the result value and zero is assigned to the imaginary part. If y is not specified and x is complex, the result value is DCPLX(REAL(x), AIMAG(x)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

DCMPLX(x, y) has the complex value whose real part is REAL(x, kind=8) and whose imaginary part is REAL(y, kind=8).

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 in Error Messages.

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 NODEDECLARE

General Compiler Directives: DECLARE generates warnings for variables that have been used but have not been declared (like the IMPLICIT NONE statement). NODEDECLARE (the default) disables these warnings.

Syntax

\texttt{\textbackslash cDEC$ DECLARE}
\texttt{\textbackslash cDEC$ NODEDECLARE}
DECLARE and NODEDECLARE

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$DECLARE and !MS$NODECLAIM

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**, **ENCOD**E

Examples

In the following example, the **DECODE** statement translates the 12 characters in \( A \) to integer form (as specified by the **FORMAT** statement):

```plaintext
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 \( K \):

\[
\begin{align*}
K(1) &= 1234 \\
K(2) &= 5678 \\
K(3) &= 9012
\end{align*}
\]

**DEFAULT** (**TU*X** only)

**Parallel Directive Clause:** Lets you specify a scope for all variables in the lexical extent of a parallel region.

**Syntax**

```
DEFAULT ( PRIVATE | SHARED | NONE )
```

The specifications have the following effects:

- **PRIVATE** - Makes all named objects in the lexical extent of the parallel...
region, including common block variables but excluding \texttt{THREADPRIVATE} (or \texttt{TASKCOMMON}) variables, private to a thread as if you explicitly listed each variable in a \texttt{PRIVATE} clause.

- \texttt{SHARED} - Makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if you explicitly listed each variable in a \texttt{SHARED} clause. If you do not specify a \texttt{DEFAULT} clause, this is the default.

- \texttt{NONE} - Specifies that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, you must specify the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION property of each variable you use in the lexical extent of the parallel region.

You can specify only one \texttt{DEFAULT} clause in a \texttt{PARALLEL} directive. You can exclude variables from a defined default by using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clauses.

Variables in \texttt{THREADPRIVATE} (or \texttt{TASKCOMMON}) common blocks are not affected by this clause.

\section*{DEFINE and UNDEFINE}

\textbf{General Compiler Directives:} \texttt{DEFINE} creates a symbolic variable whose existence or value can be tested during conditional compilation. \texttt{UNDEFINE} removes a defined symbol.

\section*{Syntax}

\begin{verbatim}
cDEC$ DEFINE name [ = val]
cDEC$ UNDEFINE name
\end{verbatim}

\texttt{c} Is one of the following: C (or c), !, or *. (See \texttt{Syntax Rules for General Directives}.)

\texttt{name} Is the name of the variable.

\texttt{val} (Input) INTEGER(4). The value assigned to \texttt{name}.

\section*{Rules and Behavior}

\texttt{DEFINE} and \texttt{UNDEFINE} create and remove symbols for use with the \texttt{IF} (or \texttt{IF}}
**DEFINE** and **UNDEFINE** 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**

```fortran
!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**

```fortran
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**

```plaintext
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**

```plaintext
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**
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 **READs** 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:
Suppose the following statement is specified:

```plaintext
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**

```plaintext
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**

```plaintext
USE DFLIB
LOGICAL(4) result
CHARACTER(25) str
str = 'Add to EDIT Menu'  ! 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**

```fortran
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**

```fortran
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
ENDIF
count = DELFILESQQ(file)
WRITE(*,*) "Deleted ", count, ", files."
```
Derived Type

Statement: Specifies the name of a user-defined type and the types of its components.

Syntax

```plaintext
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:

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

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

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

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

component
Is the name of the component being defined.

a-spec
Is an optional array specification, enclosed in parentheses. If POINTER is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression.

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

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

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

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

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

Rules and Behavior

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

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

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.
Two data entities have the same type if they are both declared to be of the same derived type (the derived-type definition can be accessed from a module or a host scoping unit).

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

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

**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**

```fortran
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`:

```fortran
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:

```fortran
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.

The following table lists the Compaq Fortran defaults for argument passing, and the allowed uses of %DESCR:

<table>
<thead>
<tr>
<th>Actual Argument Data Type</th>
<th>Default</th>
<th>%DESCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Integer</td>
<td>REF</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The %DESCR, %VAL, and %REF functions override related cDEC$ ATTRIBUTE settings.

See Also: CALL, %VAL, %REF

DFLOAT

<table>
<thead>
<tr>
<th>Data Type</th>
<th>REF</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX(8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX(16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hollerith</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Name:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure Name:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 VMS, U*X
2 On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, a character argument is passed by address and hidden length.
3 In Compaq Fortran record structures
Elemental Intrinsic Function (Generic): Converts an integer to double precision type.

Syntax

\[
\text{result} = \text{DFLOAT} (a)
\]

\(a\) (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.

<table>
<thead>
<tr>
<th>Specific Name (^1)</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFLOTI</td>
<td>INTEGER(2)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>DFLOTJ</td>
<td>INTEGER(4)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>DFLOTK (^2)</td>
<td>INTEGER(8)</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

\(^1\) 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), \( \text{DIGITS}(x) \) has the value 24.

**DIM**

**Elemental Intrinsic Function (Generic):** Returns the difference between two numbers (if the difference is positive).

**Syntax**

\[
\text{result} = \text{DIM} \ (x, \ y)
\]

\( x \)  
(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.
DIM

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
<td></td>
</tr>
<tr>
<td>IIDIM</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>IDIM ¹</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIDIM ²</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>DIM</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DDIM</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QDIM ³</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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
do
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:

\[
\text{type, [att-\text{ls},] DIMENSION (a-spec) [, att-\text{ls}] :: a[(a-spec)] [, a[(a-spec)]]]} \ldots
\]

Statement:

\[
\text{DIMENSION [::] a(a-spec) [, a(a-spec)]} \ldots
\]

\text{type}

Is a data type specifier.

\text{att-\text{ls}}

Is an optional list of attribute specifiers.

\text{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.

\text{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.

<table>
<thead>
<tr>
<th>Array element</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(1,0)</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>a(2,0)</td>
<td>1002</td>
<td>2</td>
</tr>
<tr>
<td>a(1,1)</td>
<td>1004</td>
<td>3</td>
</tr>
<tr>
<td>a(2,1)</td>
<td>1006</td>
<td>4</td>
</tr>
<tr>
<td>a(1,2)</td>
<td>1008</td>
<td>5</td>
</tr>
<tr>
<td>a(2,2)</td>
<td>100A</td>
<td>6</td>
</tr>
</tbody>
</table>

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:

```
DISPLAYCURSOR

Graphics Function: Controls cursor visibility.

Module: USE DFLIB

Syntax

```fortran
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

See Also: SETTEXTCURSOR

DLGEXIT

Run-Time Subroutine: Closes an open dialog box.

Module: USE DFLOGM

Syntax

```fortran
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**

```fortran
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE DFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
   CALL DLGEXIT (dlg)
```

**DLGFLUSH**

**Run-Time Subroutine:** Updates the display of a dialog box.

**Module:** USE DFLOGM

**Syntax**

```fortran
CALL DLGFLUSH ( dlg [, flushall] )
```

### Parameters

- **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.

- **flushall**
  (Input; optional) Logical. If .FALSE. (the default), then only the controls that the dialog routines have marked as changed are updated. If .TRUE., all controls are updated with the state of the controls as known by the dialog routines. Normally, you would not set **flushall** to .TRUE..

When your application calls **DLGSET** to change a property of a control in a dialog box, the change is not immediately reflected in the displayed dialog box.
Changes are applied when the dialog box is first displayed, and then after every dialog callback to the user's code.

This design expects that, after a call to `DLGMODAL` or `DLGMODELESS`, every call to `DLGSET` will be made from within a callback routine, and that the callback routine finishes quickly. This is true most of the time and so is not an issue.

However, there may be cases where you want to change a control outside of a dialog callback, or from within a loop in a dialog callback. In these cases, `DLGFLUSH` is required, but not always sufficient, to update the dialog display. `DLGFLUSH` sends pending Window messages to the dialog box and the controls that it contains. However, many display changes do not appear until after the program reads and processes the messages that Windows has sent to the program as the result of the messages sent by `DLGFLUSH`. A loop that processes the pending messages is often required; for example:

```fortran
use DFWINTY
use USER32

logical lNotQuit, lret
integer iret
TYPE (T_MSG) mesg

lNotQuit = .TRUE.
do while (lNotQuit .AND. (PeekMessage(mesg, 0, 0, 0, PM_NOREMOVE) <> 0))
   lNotQuit = GetMessage(mesg, NULL, 0, 0)
   if (lNotQuit) then
      if (DLGISDLGMESSAGE(mesg) .EQV. .FALSE) then
         lret = TranslateMessage(mesg)
         iret = DispatchMessage(mesg)
      end if
   end if
end do
```

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** `DLGINIT`, `DLGMODAL`, `DLGMODELESS`, `DLGSET`, `DLGSETSUB`

**DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR**

**Run-Time Functions:** Retrieve the state of the dialog control variable.

**Module:** USE DFLOGM

**Syntax**

```fortran
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 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 **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**

```fortran
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
   ...
```

**Run-Time Functions**: Determine whether the specified message is intended for one of the currently displayed modeless dialog boxes, or a specific dialog box.

**Module**: USE DFLOGM

**Syntax**

```fortran
result = DLGISDLGMESSAGE (msg)
result = DLGISDLGMESSAGEWITHDLG (msg, dlg)
```

`msg`
(Input) Derived type T_MSG. Contains a Windows message.

`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 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.

**DLGISDLGMESSAGE** must be called in the message loop of Windows applications that display a modeless dialog box using **DLGMODELESS**. **DLGISDGMESSAGE** 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.

**DLGISDLGMESSAGEWITHDLG** specifies a particular dialog box to check. Use **DLGISDLGMESSAGEWITHDLG** when the message loop is in a main application and the currently active modeless dialog box was created by a DLL.

Compatibility

WINDOWS

See Also: [DLGMODELESS, Using a Modeless Dialog Routine](#)

Example

```fortran
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 messages
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
```
Run-Time Functions: Display a dialog box and process user control selections made within the box.

Module: USE DFLOGM

Syntax

\[
\text{result} = \text{DLGMODAL} (\text{dlg}) \\
\text{result} = \text{DLGMODALWITHPARENT} (\text{dlg}, \text{hwndParent})
\]

\text{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.

\text{hwndParent}
(Input) Integer. Specifies the parent window for the dialog box. If omitted, the value is determined in this order:

1. If \text{DLGMODAL} is called from the callback of a modal or modeless dialog box, then that dialog box is the parent window.
2. If it is a QuickWin or Standard Graphics application, then the frame window is the parent window.
3. The Windows desktop window is the parent window.

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, \text{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. \text{DLGMODAL} does not return a value until the dialog box is exited.

The default return value for \text{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.

Use DLGMODALWITHPARENT when you want the parent window to be other than the default value (see hwndParent above). In particular, in an SDI or MDI Windows application, you may want the parent window to be the main application window. The parent window is disabled for user input while the modal dialog box is displayed.

**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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW_HIDE</td>
<td>Hides the dialog box.</td>
</tr>
<tr>
<td>SW_MINIMIZE</td>
<td>Minimizes the dialog box.</td>
</tr>
<tr>
<td>SW_RESTORE</td>
<td>Activates and displays the dialog box. If the dialog box is minimized or maximized, Windows restores it to its original size and position.</td>
</tr>
<tr>
<td>SW_SHOW</td>
<td>Activates the dialog box and displays it in its current size and position.</td>
</tr>
<tr>
<td>SW_SHOWMAXIMIZED</td>
<td>Activates the dialog box and displays it as a maximized window.</td>
</tr>
<tr>
<td>SW_SHOWMINIMIZED</td>
<td>Activates the dialog box and displays it as an icon.</td>
</tr>
<tr>
<td>SW_SHOWMINNOACTIVE</td>
<td>Displays the dialog box as an icon. The window that is currently active remains active.</td>
</tr>
<tr>
<td>SW_SHOWNA</td>
<td>Displays the dialog box in its current state. The window that is currently active remains active.</td>
</tr>
<tr>
<td>SW_SHOWNOACTIVATE</td>
<td>Displays the dialog box in its most recent size and position. The window that is currently active remains active.</td>
</tr>
<tr>
<td>SW_SHOWNORMAL</td>
<td>Activates and displays the dialog box. If the dialog box is minimized or maximized, Windows restores it to its original size and position.</td>
</tr>
</tbody>
</table>

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.

**DLGMODELESS** is typically 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.

**DLGMODELESS** also can be used in a Console, DLL, or LIB project. However, the requirements remain that the application must contain a message loop and must call **DLGISDLGMESSAGE** for each message. For an example of calling **DLGMODELESS** in a DLL project, see the Dllprgrs sample in the ...\SAMPLES\DIALOG folder.

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 CONSOLE DLL LIB

**See Also:** **DLGSETSUB**, **DLGINIT**, **DLGEXIT**, **DLGISDLGMESSAGE**, Using a Modeless Dialog Routine

**Example**

```fortran
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
integer          iret

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
  iret = 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).

\textit{value}  
(Input) Integer, logical, or character. The value of the control's variable.

\textit{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.

\textbf{Results:}

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the \texttt{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 \textit{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 \textit{Programmer's Guide}.

You can use \texttt{DLGSET} to set any control variable. You can also use \texttt{DLGSETINT} to set an integer variable, or \texttt{DLGSETLOG} and \texttt{DLGSETCHAR} to set logical and character values, respectively. If you use \texttt{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 \texttt{DLGSET} functions return .FALSE..

Calling \texttt{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 \texttt{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 \texttt{DLGSET} is executed.

If two or more controls have the same \textit{controlid}, you cannot use these controls in a \texttt{DLGSET} operation. In this case the function returns .FALSE..

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{DLGSETSUB}, \texttt{DLGGET}, \textit{Using Dialogs}, \textit{Dialog Functions}, and \textit{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)
...

DLGSETCTRLEVENTHANDLER

Run-Time Function: Assigns user-written event handlers to ActiveX™ controls in a dialog box.

Module: USE DFLOGM

Syntax

result = DLGSETCTRLEVENTHANDLER ( dlg, controlid, handler, dispid [, iid] )

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.

handler
(Input) EXTERNAL. Name of the routine to be called when the event occurs.

dispid
(Input) Integer. Specifies the member id of the method in the event interface that identifies the event.

iid
(Input; optional) Derived type GUID, which is defined in the DFWINTY module. Specifies the interface identifier of the source (event) interface. If omitted, the default source interface of the ActiveX control is used.

Results:
The result type is INTEGER(4). The result is an HRESULT describing the status of the operation.

When the ActiveX control event occurs, the handler associated with the event is called. You call `DLGSETCTRLEVENTHANDLER` to specify the handler to be called.

The events supported by an ActiveX control and the interfaces of the handlers are determined by the ActiveX control.

You can find this information in one of the following ways:

- By reading the documentation of the ActiveX control.
- By using a tool that lets you examine the type information of the ActiveX control, The OLE-COM Object Viewer in the Visual Fortran folder is one such tool.
- By using the Fortran Module Wizard to generate a module that contains Fortran interfaces to the ActiveX control, and examining the generated module.

The handler that you define in your application must have the interface that the ActiveX control expects, including calling convention and parameter passing mechanisms. Otherwise, your application will likely crash in unexpected ways because of the application's stack getting corrupted.

Note that an object is always the first parameter in an event handler. This object value is a pointer to the control's source (event) interface, not the IDispatch pointer of the control. You can use `DLGGET` with the DLG_IDISPATCH index to retrieve the control's IDispatch pointer.

For more information, see Using ActiveX Controls in the Programmer's Guide.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** `DLGINIT`, `DLGGET`, `DLGMODAL`, `DLGMODELESS`, `DLGSETSUB`

**Example**

```fortran
ret = DlgSetCtrlEventHandler(      &
   dlg,                         &
   IDC_ACTIVEMOVIECONTROL1,     & ! Identifies the control
   ReadyStateChange,            & ! Name of the event handling routine
   -609,                        & ! Member id of the ActiveMovie's
   & ! control ReadyStateChange event.
```
**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**

```fortran
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**

\[
\text{result} = \text{DLGSETSUB} (\text{dlg, controlid, value } [, \text{ index}])
\]

\textit{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.

\textit{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.

\textit{value}
(Input) EXTERNAL. Name of the routine to be called when the callback event occurs.

\textit{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 two callbacktypes, DLG_INIT and DLG_SIZECHANGE. The DLG_INIT
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. DLG_SIZECHANGE is called when the size of a
dialog is changed.

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

DLGSETTITLE

Run-Time Subroutine: Sets the title of a dialog box.

Module: USE DFLOGM

Syntax

CALL DLGSETTITLE (dlg, title)


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.

title
(Input) Character*(*) . Specifies text to be the title of the dialog box.

Use this routine when you want to specify the title for a dialog box.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB
See Also: **DLGINIT, DLGMODAL, DLGMODELESS**

Example

USE DFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG) mydialog
LOGICAL retlog
...
retlog = DLGINIT(IDD_mydlg, mydialog)
...
CALL DLGSETTITLE(mydialog, "New Title")
...

**DLGUNINIT**

**Run-Time Subroutine:** Deallocates memory associated with an initialized dialog.

**Module:** USE DFLOGM

**Syntax**

```fortran
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**

\[
\begin{align*}
&\text{[ name:]} \text{ DO } [\text{label[, ] }] [\text{loop-control}] \\
&\text{block} \\
&[\text{label}] \text{ term-stmt}
\end{align*}
\]

**Nonblock Form**

**DO** \(\text{label[,]} \ [\text{loop-control}]\)

- **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 \texttt{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:

- \texttt{CONTINUE} (allowed if it is a shared terminal statement)
- \texttt{CYCLE}
- \texttt{END} (for a program or subprogram)
- \texttt{EXIT}
- \texttt{GO TO} (unconditional or assigned)
- Arithmetic \texttt{IF}
- \texttt{RETURN}
- \texttt{STOP}

The nonblock DO construct is an \textit{obsolescent feature} in Fortran 95 and Fortran 90.

\textbf{Compatibility}

\texttt{CONSOLE} \texttt{STANDARD} \texttt{GRAPHICS} \texttt{QUICKWIN} \texttt{GRAPHICS} \texttt{WINDOWS} \texttt{DLL} \texttt{LIB}

\textbf{See Also:} \texttt{CONTINUE}, \texttt{CYCLE}, \texttt{EXIT}, \texttt{DO WHILE}, \texttt{Execution Control}, \texttt{DO Constructs}

\textbf{Examples}

The following example shows a simple block DO construct (contains no iteration count or \texttt{DO WHILE} statement):

\begin{verbatim}
  DO
    READ *, N
    IF (N == 0) STOP
    CALL SUBN
  END DO
\end{verbatim}

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:

\begin{verbatim}
  LOOP_1: DO I = 1, N
           A(I) = C * B(I)
  END DO LOOP_1
\end{verbatim}
The following example shows a nonblock **DO** construct with a shared terminal statement:

```plaintext
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:

```plaintext
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):

```plaintext
DO j = 1, 10
   WRITE (*, '(I5)') j
END DO
WRITE (*, '(I5)') j
```

**DO Directive (TU*X only)**

**OpenMP Parallel Compiler Directive:** Specifies that the iterations of the immediately following **DO** loop must be executed in parallel.

**Syntax**

```plaintext
c$OMP DO [clause[[, clause] ... ]
do_loop
[ [c$OMP END DO [NOWAIT]]]
c
   Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

   clause
   Is one of the following:

   - FIRSTPRIVATE (list)
```
- **LASTPRIVATE (list)**

- **ORDERED**

  Must be used if ordered sections are contained in the dynamic extent of the **DO** directive. For more information about ordered sections, see the **ORDERED directive**.

- **PRIVATE (list)**

- **REDUCTION( operator | intrinsic : list)**

- **SCHEDULE (type[, chunk])**

  Specifies how iterations of the **DO** loop are divided among the threads of the team. **chunk** must be a scalar integer expression. The following four types are permitted, three of which allow the optional parameter **chunk**:

<table>
<thead>
<tr>
<th>Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Divides iterations into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins.</td>
</tr>
<tr>
<td></td>
<td>If <strong>chunk</strong> is specified, iterations are divided into pieces of a size specified by <strong>chunk</strong>. The pieces are statically dispatched to threads in the team in a round-robin fashion in the order of the thread number.</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Can be used to get a set of iterations dynamically. It defaults to 1 unless <strong>chunk</strong> is specified.</td>
</tr>
<tr>
<td></td>
<td>If <strong>chunk</strong> is specified, the iterations are broken into pieces of a size specified by <strong>chunk</strong>. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations.</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Can be used to specify a minimum number of iterations. It defaults to 1 unless <strong>chunk</strong> is specified.</td>
</tr>
<tr>
<td></td>
<td>If <strong>chunk</strong> is specified, the chunksize is reduced exponentially with each succeeding dispatch. The <strong>chunk</strong> specifies the minimum number of iterations to dispatch each time. If there are less than <strong>chunk</strong></td>
</tr>
</tbody>
</table>
If the SCHEDULE clause is not used, the default schedule type is STATIC.

**do_loop**

Is a **DO** iteration (an iterative **DO** loop). It cannot be a **DO WHILE** or a **DO** loop without loop control. The **DO** loop iteration variable must be of type integer.

The iterations of the **DO** loop are distributed across the existing team of threads. The values of the loop control parameters of the **DO** loop associated with a **DO** directive must be the same for all the threads in the team.

You cannot branch out of a **DO** loop associated with a **DO** directive.

**Rules and Behavior**

If used, the **END DO** directive must appear immediately after the end of the loop. If you do not specify an **END DO** directive, an **END DO** directive is assumed at the end of the **DO** loop.

If you specify NOWAIT in the **END DO** directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instruction following the loop without waiting for the other members of the team to finish the **DO** directive.

Parallel **DO** loop control variables are block-level entities within the **DO** loop. If the loop control variable also appears in the LASTPRIVATE list of the parallel **DO**, it is copied out to a variable of the same name in the enclosing **PARALLEL** region. The variable in the enclosing **PARALLEL** region must be SHARED if it is specified in the LASTPRIVATE list of a **DO** directive.

Only a single SCHEDULE clause and ORDERED clause can appear in a **DO** directive.

**DO** directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

**See Also:** Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API
Examples

In the following example, the loop iteration variable is private by default, and it is not necessary to explicitly declare it. The **END DO** directive is optional:

```fortran
C$OMP PARALLEL
C$OMP DO
   DO I=1,N
      B(I) = (A(I) + A(I-1)) / 2.0
   END DO
C$OMP END DO
C$OMP END PARALLEL
```

If there are multiple independent loops within a parallel region, you can use the **NOWAIT** keyword to avoid the implied **BARRIER** at the end of the **DO** directive, as follows:

```fortran
C$OMP PARALLEL
C$OMP DO
   DO I=2,N
      B(I) = (A(I) + A(I-1)) / 2.0
   END DO
C$OMP END DO NOWAIT
C$OMP DO
   DO I=1,M
      Y(I) = SQRT(Z(I))
   END DO
C$OMP END DO NOWAIT
C$OMP END PARALLEL
```

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a **LASTPRIVATE** clause so that the values of the variables are the same as when the loop is executed sequentially, as follows:

```fortran
C$OMP PARALLEL
C$OMP DO LASTPRIVATE(I)
   DO I=1,N
      A(I) = B(I) + C(I)
   END DO
C$OMP END PARALLEL
CALL REVERSE(I)
```

In this case, the value of I at the end of the parallel region equals N+1, as in the sequential case.

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```fortran
C$OMP DO ORDERED SCHEDULE(DYNAMIC)
```
DO Directive (TU*X only)

DO I=LB,UB,ST
   CALL WORK(I)
END DO
...
SUBROUTINE WORK(K)
c$OMP ORDERED
   WRITE(*,*) K
   c$OMP END ORDERED

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:

<table>
<thead>
<tr>
<th>Required</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO WHILE (I .GT. J)</td>
<td>DO 10 WHILE (I .GT. J)</td>
</tr>
<tr>
<td>ARRAY(I,J) = 1.0</td>
<td>ARRAY(I,J) = 1.0</td>
</tr>
<tr>
<td>I = I - 1</td>
<td>I = I - 1</td>
</tr>
<tr>
<td>END DO</td>
<td>10 END DO</td>
</tr>
</tbody>
</table>

The following shows another example:

```fortran
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**

```fortran
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: \((1 \times 3) + (2 \times 4) + (3 \times 5)\) = 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 higher precision product. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Syntax

```fortran
result = DPROD (x, y)
```

\( x \)
---
(Input) Must be of type REAL(4) or REAL(8).

\( y \)
---
(Input) Must have the same type and kind parameters as \( x \).

Results:

If \( x \) and \( y \) are of type REAL(4), the result type is double precision real. If \( x \) and \( y \) are of type REAL(8), the result type is REAL(16). The result value is equal to \( x \times y \).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Examples

DPROD (2.0, -4.0) has the value -8.00D0.

DPROD (5.0D0, 3.0D0) has the value 15.00Q0.

The following shows another example:

```fortran
REAL(4) e
REAL(8) d
e = 123456.7
d = 123456.7D0
! DPROD (e, e) returns 15241557546.4944
! DPROD (d, d) returns 15241556774.8899992813874268904328
```

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:

<table>
<thead>
<tr>
<th>Value of iflag</th>
<th>Selection process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The generator is restarted and the first random value is selected.</td>
</tr>
<tr>
<td>0</td>
<td>The next random number in the sequence is selected.</td>
</tr>
<tr>
<td>Otherwise</td>
<td>The generator is reseeded using iflag, restarted, and the first random value is selected.</td>
</tr>
</tbody>
</table>

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**

```fortran
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,'
```

```fortran
END SUBROUTINE
END
```
**DREAL**

**Elemental Intrinsic Function (Specific):** Converts the real part of a double complex argument to double precision type. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

**Syntax**

\[ \text{result} = \text{DREAL} (a) \]

\[ a \]

(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 2.0d0.

**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 (including Windows 2000) systems.

**Module:** USE DFPORT

**Syntax**

\[ \text{result} = \text{DTIME} (tarray) \]

\[ tarray \]

(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**

```fortran
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 this keyword in a **FUNCTION** or **SUBROUTINE** statement.

An explicit interface must be visible to the caller of an ELEMENTAL procedure.

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**, **Determining When Procedures Require Explicit Interfaces**, **Optional Arguments**

**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 \((x_1, y_1)\) and \((x_2, y_2)\). When you use \texttt{ELLIPSE\_W}, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points \((wx_1, wy_1)\) and \((wx_2, wy_2)\). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The control option given by \$\texttt{GFILLINTERIOR} is equivalent to a subsequent call to the \texttt{FLOODFILLRGB} function using the center of the ellipse as the start point and the current color (set by \texttt{SETCOLORRGB}) as the boundary color.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

**See Also:** \texttt{ARC}, \texttt{FLOODFILLRGB}, \texttt{GRSTATUS}, \texttt{LINETO}, \texttt{PIE}, \texttt{POLYGON}, \texttt{RECTANGLE}, \texttt{SETCOLORRGB}, \texttt{SETFILLMASK}

**Example**

This program draws the shape shown below.

```fortran
! 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**

See the \texttt{IF Construct}.  

**ELSE Directive**

See the \texttt{IF Directive Construct}.  

**ELSEIF Directive**

See the \texttt{IF Directive Construct}.  

\(x_1, y_1\)

\(x_2, y_2\)
ELSE IF

See the 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.
    The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.
  
  - 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" during the processing of the ELSEWHERE statement.

Compatibility
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:

```fortran
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**
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
C An END statement must be the last statement in a program
C unit:
PROGRAM MyProg
WRITE (*, '("Hello, world!")')
END

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

Rules and Behavior

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:

```plaintext
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 the 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:
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 one or more entry points within a subprogram. It is not executable and must precede any **CONTAINS** statement (if any) within the subprogram.
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

ENTRY statements can only appear in external procedures or module procedures.

An ENTRY statement must not appear in a CASE, DO, IF, FORALL, or WHERE construct, or a nonblock DO loop.

When the ENTRY statement appears in a subroutine subprogram, it is referenced by a CALL statement. When the ENTRY statement appears in a function subprogram, it is referenced by a function reference.

An entry name within a function subprogram can appear in a type declaration statement.

Within the subprogram containing the ENTRY statement, the entry name must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement, and it must not appear in an EXTERNAL or INTRINSIC statement. For example, neither of the following are valid:

(1) SUBROUTINE SUB(E)
ENTRY E
   ...

(2) SUBROUTINE SUB
EXTERNAL E
ENTRY E
An **ENTRY** statement can reference itself if the function or subroutine subprogram was defined as **RECURSIVE**.

Dummy arguments can be used in **ENTRY** statements even if they differ in order, number, type and kind parameters, and name from the dummy arguments used in the **FUNCTION**, **SUBROUTINE**, and other **ENTRY** statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement.

Dummy arguments can be referred to only in executable statements that follow the first **SUBROUTINE**, **FUNCTION**, or **ENTRY** statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced. Arguments do not retain their association from one reference of a subprogram to another.

**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
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 (Specific):** Checks whether a file is at or beyond the end-of-file record. This is a specific function that has no generic function
associated with it. It must not be passed as an actual argument.

**Syntax**

\[
\text{result} = \text{EOF} (a)
\]

\(a\)

(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..

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** [ENDFILE](#), [BACKSPACE](#), [REWIND](#)

**Example**

```fortran
! 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**

\[
\text{result} = \text{EOSHIFT} (\text{array}, \text{shift} [, \text{boundary}][, \text{dim}])
\]

*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 \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) 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 \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\). 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*):

<table>
<thead>
<tr>
<th>array Type</th>
<th>boundary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Real</td>
<td>0.0</td>
</tr>
<tr>
<td>Complex</td>
<td>((0.0, 0.0))</td>
</tr>
<tr>
<td>Logical</td>
<td>false</td>
</tr>
<tr>
<td>Character(len)</td>
<td>len blanks</td>
</tr>
</tbody>
</table>

*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 \texttt{array} has rank greater than one, each section \((s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n)\) of the result is shifted as follows:

- By the value of \texttt{shift}, if \texttt{shift} is scalar
- According to the corresponding value in \texttt{shift}(s_1, s_2, \ldots, s_{\text{dim}-1}, s_{\text{dim}+1}, \ldots, s_n), if \texttt{shift} is an array

If an element is shifted off one end of a section, the \texttt{boundary} value is placed at the other end of the section.

The value of \texttt{shift} determines the amount and direction of the end-off shift. A positive \texttt{shift} value causes a shift to the left (in rows) or up (in columns). A negative \texttt{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:** \texttt{CSHIFT, ISHFT, ISHFTC, TRANSPOSE}

**Examples**

\(V\) is the array \((1, 2, 3, 4, 5, 6)\).

\[
\text{EOSHIFT} (V, \text{SHIFT}=2) \text{ shifts the elements in } \text{\texttt{V}} \text{ to the \texttt{left} by } 2 \text{ positions, producing the value } (3, 4, 5, 6, 0, 0). \text{ 1 and 2 are shifted off the beginning and two elements with the default \texttt{BOUNDARY} value are placed at the end.}
\]

\[
\text{EOSHIFT} (V, \text{SHIFT}=-3, \text{BOUNDARY}=99) \text{ shifts the elements in } \text{\texttt{V}} \text{ to the \texttt{right} by } 3 \text{ positions, producing the value } (99, 99, 99, 1, 2, 3). \text{ 4, 5, and 6 are shifted off the end and three elements with \texttt{BOUNDARY} value 99 are placed at the beginning.}
\]

\(M\) is the array

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}.
\]

\[
\text{EOSHIFT} (M, \text{SHIFT} = 1, \text{BOUNDARY} = '\texttt{*}'\), \text{DIM} = 2) \text{ produces the result }
\]

\[
\begin{bmatrix}
2 & 3 & * \\
5 & 6 & * \\
8 & 9 & * \\
\end{bmatrix}.
\]
Each element in rows 1, 2, and 3 is shifted to the left by 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

\[
\begin{bmatrix}
0 & 0 & 0 \\
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}.
\]

Each element in columns 1, 2, and 3 is shifted down by 1 position. This causes the last element in each column to be shifted off the end and the BOUNDARY value to be placed at the beginning.

**EOSHIFT** (M, SHIFT = (/1, -1, 0/), BOUNDARY = (/ '*', '?', '/' /), DIM = 2) produces the result

\[
\begin{bmatrix}
2 & 3 & * \\
? & 4 & 5 \\
7 & 8 & 9
\end{bmatrix}.
\]

Each element in row 1 is shifted to the left by 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:

```fortran
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 a scalar of the same type and kind parameters as x. The result has the value $b^{1-p}$. Parameters $b$ and $p$ are defined in 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:

<table>
<thead>
<tr>
<th>Type of Object</th>
<th>Type of Associated Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrinsic numeric ¹ or numeric sequence</td>
<td>Can be of any of these types</td>
</tr>
<tr>
<td>Default character or character sequence</td>
<td>Can be of either of these types ²</td>
</tr>
<tr>
<td>Any other intrinsic type</td>
<td>Must have the same type and kind parameters</td>
</tr>
<tr>
<td>Any other sequence type</td>
<td>Must have the same type</td>
</tr>
</tbody>
</table>

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:
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):

```fortran
REAL, DIMENSION(2), :: X
REAL :: Y
EQUIVALENCE (X(1), Y), (X(2), Y)
```

The following EQUIVALENCE statement is invalid because A(1) and A(2) will not be consecutive:

```fortran
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.

```fortran
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).

```fortran
CHARACTER KEY*16, STAR*10
EQUIVALENCE (KEY, STAR)
```

The following shows another example:

```fortran
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:

```fortran
CHARACTER (LEN = 4) :: a, b
CHARACTER (LEN = 3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```
This causes the following alignment:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a(1:1)</td>
<td>a(2:2)</td>
<td>a(3:3)</td>
<td>a(4:4)</td>
<td>b(1:1)</td>
<td>b(2:2)</td>
<td>b(3:3)</td>
</tr>
<tr>
<td></td>
<td>c(1)(1:1)</td>
<td>c(1)(2:2)</td>
<td>c(1)(3:3)</td>
<td>c(2)(1:1)</td>
<td>c(2)(2:2)</td>
<td>c(2)(3:3)</td>
<td>b(4:4)</td>
</tr>
</tbody>
</table>

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 Compaq Fortran Run-Time Library error number that occurred during program execution. (For a listing of error numbers, see Visual Fortran Run-Time Errors in Error Messages.)
  - 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 one of the following:
    - On Windows NT (including Windows 2000) and Windows 9* systems, it is the value returned by `GETLASTERROR` at the time of the error.
    - On OpenVMS systems, it is an RMS STS value.
    - On Tru64 UNIX and Linux systems, it is an `errno` value. (See `errno(2)`.)

- **stat**
  - (Optional) Is an integer variable or array element that stores a status value that occurred during program execution. This value is one of the following:
    - On OpenVMS systems, it is an RMS STV value.
    - On Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux systems, 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 (including Windows 2000) 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

\[
\text{result} = \text{EXP} (x)
\]

\[x\]

(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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DEXP</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QEXP 1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>CEXP 2</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
</tr>
<tr>
<td>CDEXP 3</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
</tr>
<tr>
<td>CQEXP 1</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
</tr>
</tbody>
</table>

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, sizeof, time, rate
sizei = 10000.0
time = 40.5
rate = 0.0875
sizeof = sizei * EXP (rate * time)
WRITE (*, 100) sizeof
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**

\[ \text{result} = \text{EXPONENT} \left( x \right) \]

\( x \) (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
EXPONENT

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:

\text{type}, \text{[att-ls,]} \text{EXTERNAL} \text{[, att-ls]} :: \text{ex-pro} \text{[, ex-pro]} ... 

Statement:

\text{EXTERNAL} \text{ex-pro} \text{[, ex-pro]} ... 

\text{type}
Is a data type specifier.

\text{att-ls}
Is an optional list of attribute specifiers.

\text{ex-pro}
Is the name of an external (user-supplied) procedure or dummy procedure.

Rules and Behavior

In a type declaration statement, only \textit{functions} can be declared EXTERNAL. However, you can use the \textbf{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 \texttt{SIN} is declared with the EXTERNAL attribute, all subsequent references to \texttt{SIN} are to a user-supplied function named \texttt{SIN}, not to the intrinsic function of the same name.

You can include the name of a block data program unit in the \textbf{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, FORTRAN-66 Interpretation of the External Statement

**Examples**

The following example shows type declaration statements specifying the EXTERNAL attribute:

```fortran
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:

```fortran
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:

```fortran
CALL SUBR(A, FUNC(B), C)
```

The following shows another example:

```fortran
EXTERNAL MyFunc, MySub
C   MyFunc and MySub are arguments to Calc
  CALL Calc (MyFunc, MySub)
C   Example of a user-defined function replacing an intrinsic
C intrinsic
  EXTERNAL SIN
  x = SIN (a, 4.2, 37)
```
**Portability Function and Subroutine:** Returns the current date and time as an ASCII string.

**Module:** USE DFPORT

**Subroutine Syntax**

```fortran
CALL FDATE ( [string] )
```

**Function Syntax**

```fortran
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:

```fortran
CHARACTER*24 FDATE
```

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** [DATE_AND_TIME](#)

**Example**

```fortran
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
Module: USE DFPORT

Syntax

\[
\text{result} = \text{FGETC}(lunit, \ char)
\]

\text{result} \quad \text{(Output)} \quad \text{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:}

\begin{itemize}
  \item \text{EINVAL: The specified unit is invalid (either not already open, or an invalid unit number).}
\end{itemize}

If you use \text{WRITE, READ}, or any other Fortran I/O statements with \text{lunit}, be sure to read \text{Input and Output With Portability Routines} in the \text{Programmer's Guide}.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: GETCHARQQ, READ

Example

\begin{verbatim}
USE dfport
CHARACTER inchar
INTEGER istatus
istatus = FGETC(5,inchar)
PRINT *, inchar
END
\end{verbatim}

FIND

\text{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

**FIRSTPRIVATE** *(TU*X only)*

**Parallel Directive Clause:** Provides a superset of the functionality provided by the PRIVATE clause; objects are declared PRIVATE and they are initialized with certain values.

**Syntax**

**FIRSTPRIVATE** *(list)*

*list*
Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).
Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, private (local) copies of each variable in the different threads are initialized to the value the variable had before the parallel region started.

**FIXEDFORMLINESIZE**

**General Compiler Directive:** Sets the line length for fixed-form Fortran source code.

**Syntax**

\[
\text{cDEC$ FIXEDFORMLINESIZE:}\{72 | 80 | 132\}
\]

\[
\text{c}
\]

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**

\[
\text{cDEC$ NOFREEFORM}
\]
\[
\text{cDEC$ FIXEDFORMLINESIZE:132}
\]
\[
\text{WRITE (*,*) 'Sentence that goes beyond the 72nd column without continuation.'}
\]

**FLOAT**

**Elemental Intrinsic Function (Generic):** Converts an integer to REAL(4). For
FLOODFILL, FLOODFILL_W

Graphics Function: Fills an area using the current color index and fill mask.

Module: USE DFLIB

Syntax

\[
\begin{align*}
\text{result} &= \text{FLOODFILL} \left( x, y, \text{bcolor} \right) \\
\text{result} &= \text{FLOODFILL_W} \left( wx, wy, \text{bcolor} \right)
\end{align*}
\]

\(x, y\)  
(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
FLOODFILL, FLOODFILL_W

**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**

\[
result = \text{FLOODFILLRGB} \ (x, y, color) \\
result = \text{FLOODFILLRGB}_W \ (wx, wy, color)
\]

\[
x, y \\
\text{(Input) INTEGER(2). Viewport coordinates for fill starting point.}
\]

\[
wx, wy \\
\text{(Input) REAL(8). Window coordinates for fill starting point.}
\]

\[
\text{color} \\
\text{(Input) INTEGER(4). RGB value of the boundary color.}
\]

**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 \textit{color}.

\textbf{Compatibility}

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

\textbf{See Also:} ELLIPSE, FLOODFILL, GETCOLORRGB, GETFILLMASK, GRSTATUS, PIE, SETCLIPRGN, SETCOLORRGB, SETFILLMASK

\textbf{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

\textbf{FLOOR}

\textbf{Elemental Intrinsic Function (Generic):} Returns the greatest integer less than or equal to its argument.

\textbf{Syntax}

\[
\text{result} = \text{FLOOR} \left( a \ [, \ kind] \right)
\]

\(a\)  
(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

**FLUSH Directive** *(TU*X only)*
OpenMP Parallel Compiler Directive: Identifies synchronization points at which the implementation must provide a consistent view of memory.

Syntax

```
c$OMP FLUSH [(list)]
```

- `c` is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

- `list` is the name of one or more variables to be flushed. Names must be separated by commas.

Rules and Behavior

The **FLUSH** directive must appear at the precise point in the code at which the synchronization is required. To avoid flushing all variables, specify a `list`.

Thread-visible variables are written back to memory at the point at which this directive appears. Modifications to thread-visible variables are visible to all threads after this point. Subsequent reads of thread-visible variables fetch the latest copy of the data.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

The **FLUSH** directive is implied for the following directives (unless the NOWAIT keyword is used):

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED
See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples

The following example uses the **FLUSH** directive for point-to-point synchronization between pairs of threads:

```c
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
  IAM = OMP_GET_THREAD_NUM()
  ISYNC(IAM) = 0
C$OMP BARRIER
  CALL WORK()
  C I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
  ISYNC(IAM) = 1
C$OMP FLUSH(ISYNC)
  C WAIT TILL NEIGHBOR IS DONE
  DO WHILE (ISYNC(NEIGH) .EQ. 0)
    C$OMP FLUSH(ISYNC)
    END DO
C$OMP END PARALLEL
```

**FOCUSQQ**

QuickWin Function: Sets focus to the window with the specified unit number.

Module: USE DFLIB

Syntax

```v
result = FOCUSQQ (iunit)
```

`iunit`

(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
A window can be made active without getting the focus by calling SETACTIVE.

A window has focus when it is given the focus by FOCUS, 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, SETACTIVE, INQFOCUS.

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 in the Programmer's Guide.

Example
Consider the following C code:

```c
void __stdcall for_check_flawed_pentium ( void );
for_check_flawed_pentium ( );
```

Consider the following Fortran code that checks for the divide flaw:

```fortran
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**

```fortran
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:

```fortran
USE DFLIB

INTEGER*4 FPE_FLAGS
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 in the Programmer's Guide.

To initialize the Fortran run-time environment, use FOR_RTL_INIT_.

**Example**

Consider the following C code:

```c
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:

```c
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**

```c
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:

```c
USE DFLIB

INTEGER*4 OLD_FPE_FLAGS, NEW_FPE_FLAGS
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

\[
\text{result} = \text{FOR\_SET\_REENTRANCY}(\ mode \)
\]

\textit{mode}
Must be of type INTEGER(4) and contain one of the following options:

- \textbf{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.

- \textbf{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.

- \textbf{FOR\_K\_REENTRANCY\_THREADED}
  Tells the Fortran RTL to perform thread locking. This type of reentrancy should be used in multithreaded applications.

- \textbf{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 \textbf{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, \textbf{FOR\_SET\_REENTRANCY} ignores a request for thread protection (\textbf{FOR\_K\_REENTRANCY\_THREADED}) if you do not build your program with the thread-safe RTL.

Example

Consider the following:

\begin{verbatim}
PROGRAM SETREENT
USE DFLIB
\end{verbatim}
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 95/90 masked array assignment (WHERE statement and construct). It allows more general array shapes to be assigned, especially in construct form.

FORALL is a feature of Fortran 95.

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 subscripts 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-specs*.

*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:

```fortran
REAL, DIMENSION(N, N) :: A
FORALL (I=1:N) A(I, I) = 1
```

Consider the following:

```fortran
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:

```fortran
WHERE(A /= 0.0) B = 1.0 / A
```

It is also equivalent to:

```fortran
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:

```fortran
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:

```fortran
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:

```plaintext
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 \texttt{READ}, \texttt{WRITE}, or \texttt{PRINT} statement.

The following table summarizes the edit descriptors:

<table>
<thead>
<tr>
<th>Code</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A[w]</td>
<td>Transfers character or Hollerith values.</td>
</tr>
<tr>
<td>B</td>
<td>Bw[.m]</td>
<td>Transfers binary values.</td>
</tr>
<tr>
<td>D</td>
<td>Dw.d</td>
<td>Transfers real values with D exponents.</td>
</tr>
<tr>
<td>E</td>
<td>Ew.d[Ee]</td>
<td>Transfers real values with E exponents.</td>
</tr>
<tr>
<td>EN</td>
<td>ENw.d [Ee]</td>
<td>Transfers real values with engineering notation.</td>
</tr>
<tr>
<td>ES</td>
<td>ESw.d [Ee]</td>
<td>Transfers real values with scientific notation.</td>
</tr>
<tr>
<td>F</td>
<td>Fw.d</td>
<td>Transfers real values with no exponent.</td>
</tr>
<tr>
<td>G</td>
<td>Gw.d [Ee]</td>
<td>Transfers values of all intrinsic types.</td>
</tr>
<tr>
<td>I</td>
<td>Iw[.m]</td>
<td>Transfers decimal integer values.</td>
</tr>
<tr>
<td>L</td>
<td>Lw</td>
<td>Transfers logical values: on input, transfers characters; on output, transfers T or F.</td>
</tr>
<tr>
<td>O</td>
<td>Ow[.m]</td>
<td>Transfers octal values.</td>
</tr>
<tr>
<td>Z</td>
<td>Zw[.m]</td>
<td>Transfers hexadecimal values.</td>
</tr>
</tbody>
</table>
1. $w$ is the field width  
2. $m$ is the minimum number of digits that must be in the field (including zeros).  
3. $d$ is the number of digits to the right of the decimal point  
4. $E$ is the exponent field  
5. $e$ is the number of digits in the exponent

### Control Edit Descriptors

<table>
<thead>
<tr>
<th>Code</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN</td>
<td>BN</td>
<td>Ignores embedded and trailing blanks in a numeric input field.</td>
</tr>
<tr>
<td>BZ</td>
<td>BZ</td>
<td>Treats embedded and trailing blanks in a numeric input field as zeros.</td>
</tr>
<tr>
<td>P</td>
<td>kP</td>
<td>Interprets certain real numbers with a specified scale factor.</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>Returns the number of characters remaining in an input record.</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>Writes optional plus sign (+) into numeric output fields.</td>
</tr>
<tr>
<td>SS</td>
<td>SS</td>
<td>Suppresses optional plus sign (+) in numeric output fields.</td>
</tr>
<tr>
<td>T</td>
<td>Tn</td>
<td>Tabs to specified position.</td>
</tr>
<tr>
<td>TL</td>
<td>TLn</td>
<td>Tabs left the specified number of positions.</td>
</tr>
<tr>
<td>TR</td>
<td>TRn</td>
<td>Tabs right the specified number of positions.</td>
</tr>
<tr>
<td>X</td>
<td>nX</td>
<td>Skips the specified number of positions.</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>Suppresses trailing carriage return during interactive I/O.</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>Terminates format control if there are no more items in the I/O list.</td>
</tr>
<tr>
<td>/</td>
<td>[r]/</td>
<td>Terminates the current record and moves to the next record.</td>
</tr>
<tr>
<td>\</td>
<td>\</td>
<td>Continues the same record; same as $.</td>
</tr>
</tbody>
</table>

### String Edit Descriptors
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:

<table>
<thead>
<tr>
<th>Class of Argument</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signaling NaN</td>
<td>FOR_K_FP_SNAN</td>
</tr>
<tr>
<td>Quiet NaN</td>
<td>FOR_K_FP_QNAN</td>
</tr>
</tbody>
</table>

2 These delimiters can also be quotation marks (").
<table>
<thead>
<tr>
<th>Positive Infinity</th>
<th>FOR_K_FP_POS_INF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative Infinity</td>
<td>FOR_K_FP_NEG_INF</td>
</tr>
<tr>
<td>Positive Normalized Number</td>
<td>FOR_K_FP_POS_NORM</td>
</tr>
<tr>
<td>Negative Normalized Number</td>
<td>FOR_K_FP_NEG_NORM</td>
</tr>
<tr>
<td>Positive Denormalized Number</td>
<td>FOR_K_FP_POS_DENORM</td>
</tr>
<tr>
<td>Negative Denormalized Number</td>
<td>FOR_K_FP_NEG_DENORM</td>
</tr>
<tr>
<td>Positive Zero</td>
<td>FOR_K_FP_POS_ZERO</td>
</tr>
<tr>
<td>Negative Zero</td>
<td>FOR_K_FP_NEG_ZERO</td>
</tr>
</tbody>
</table>

The preceding return values are defined in file fordef.for in `\DF98\INCLUDE` on Windows NT (including Windows 2000) and Windows 9* systems; file fordef.f on Tru64 UNIX and Linux systems; and module FORSYSDEF on OpenVMS systems.

**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
endo
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
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 Windows NT (including Windows 2000) and Windows 9* systems; INTEGER(8) on OpenVMS, Tru64 UNIX, and Linux systems. 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**

FREEFORM and NOFREEFORM

**General 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
DEC$ NOFREEFORM
```

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**

```f
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:
Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, returns an error code equal to EINVAL (\texttt{lunit} is not a valid unit number, or is not open).

Mode is a bitmap consisting of an \textbf{IOR} of the following constants (the module DFPORT supplies parameters with the symbolic names given):

<table>
<thead>
<tr>
<th>\textbf{Symbolic name}</th>
<th>\textbf{Constant}</th>
<th>\textbf{Description}</th>
<th>\textbf{Notes}</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IFMT</td>
<td>O'0170000'</td>
<td>Type of file</td>
<td></td>
</tr>
<tr>
<td>S_IFDIR</td>
<td>O'0040000'</td>
<td>Directory</td>
<td></td>
</tr>
<tr>
<td>S_IFCHR</td>
<td>O'0020000'</td>
<td>Character special</td>
<td>Never set</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
<td>Description</td>
<td>Setting</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>S_IFBLK</td>
<td>O'0060000'</td>
<td>Block special</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IFREG</td>
<td>O'0100000'</td>
<td>Regular</td>
<td></td>
</tr>
<tr>
<td>S_IFLNK</td>
<td>O'0120000'</td>
<td>Symbolic link</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IFSOCK</td>
<td>O'0140000'</td>
<td>Socket</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISUID</td>
<td>O'0004000'</td>
<td>Set user ID on execution</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISGID</td>
<td>O'0002000'</td>
<td>Set group ID on execution</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISVTX</td>
<td>O'0001000'</td>
<td>Save swapped text</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>O'0000700'</td>
<td>Owner's file permissions</td>
<td></td>
</tr>
<tr>
<td>S_IRUSR, S_IREAD</td>
<td>O'0000400'</td>
<td>Owner read permission</td>
<td>Always true</td>
</tr>
<tr>
<td>S_IWUSR, S_IWRITE</td>
<td>O'0000200'</td>
<td>Owner write permission</td>
<td></td>
</tr>
<tr>
<td>S_IXUSR, S_IEXEC</td>
<td>O'0000100'</td>
<td>Owner execute permission</td>
<td>Set if S_IREAD is set</td>
</tr>
<tr>
<td>S_IRWXG</td>
<td>O'0000070'</td>
<td>Group's file permissions</td>
<td>Same as S_IRWXU</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>O'0000040'</td>
<td>Group read permission</td>
<td>Same as S_IRUSR</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>O'0000020'</td>
<td>Group write permission</td>
<td>Same as S_IWUSR</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>O'0000010'</td>
<td>Group execute permission</td>
<td>Same as S_IXUSR</td>
</tr>
<tr>
<td>S_IRWXO</td>
<td>O'0000007'</td>
<td>Other's file permissions</td>
<td>Same as S_IRWXU</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>O'0000004'</td>
<td>Other's read permission</td>
<td>Same as S_IRUSR</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>O'0000002'</td>
<td>Other write permission</td>
<td>Same as S_IWUSR</td>
</tr>
</tbody>
</table>
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**

```fortran
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**

```fortran
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**

```fortran
```

<table>
<thead>
<tr>
<th>S_ixoth</th>
<th>O'0000001'</th>
<th>Other execute permission</th>
<th>Same as S_ixusr</th>
</tr>
</thead>
</table>

| Time values are returned as number of seconds since 0:00:00 Greenwich mean time, January 1, 1970. |  |  |  |
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(DFLIB_MAXPATH) pathbuf
$\text{MAXPATH}$ 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**

```fortran
USE DFLIB

CHARACTER($\text{MAXPATH}$) buf
CHARACTER(3) drive
CHARACTER(256) dir
CHARACTER(256) name
CHARACTER(256) ext
CHARACTER(256) file

INTEGER(4) len

DO WHILE (.TRUE.)
    WRITE (*,*) ' 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 (*,*) ' Can''t get full path'
END IF

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**

[\texttt{prefix}] \textbf{FUNCTION} \textit{name} ([\texttt{d-arg-list}]) \textbf{RESULT} (\textit{r-name})

\textit{prefix}  
(Optional) Is one of the following:

\begin{itemize}
  \item \textit{keyword} \texttt{[type]}
  \item \texttt{type}\texttt{[keyword]}
\end{itemize}

\textit{type}  
Is a data type specifier.

\textit{keyword}  
Is one of the following:

\begin{tabular}{|c|l|}
  \hline
  \textbf{Keyword} & \textbf{Meaning} \\
  \hline
  \texttt{RECURSIVE} & Permissions direct recursion to occur. If a function is directly recursive and array valued, \textbf{RESULT} must also be specified. \\
  \hline
  \texttt{PURE} & Asserts that the procedure has no side effects. \\
  \hline
  \texttt{ELEMENTAL} & Restricted form of pure procedure that acts on one array element at a time. \\
  \hline
\end{tabular}

\textit{name}  
Is the name of the function. If \textbf{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 \texttt{CHARACTER}.

\textit{d-arg-list}  
(Optional) Is a list of one or more dummy arguments.

\textit{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; it is an obsolescent feature in Fortran 95.

If the length is specified as an integer constant, the value must agree with the length of the function specified in the program unit that invokes the function. If no length is specified, a length of 1 is assumed.

If the function is array-valued or a pointer, the declarations within the function must state these attributes for the function result name. The specification of the function result attributes, dummy argument attributes, and the information in the procedure heading collectively define the interface of the function.

The value of the result variable is returned by the function when it completes execution. Certain rules apply depending on whether the result is a pointer, as follows:

- If the result is a pointer, its allocation status must be determined before the function completes execution. (The function must associate a target with the pointer, or cause the pointer to be explicitly disassociated from a target.)

  The shape of the value returned by the function is determined by the shape of the result variable when the function completes execution.

- If the result is not a pointer, its value must be defined before the function completes execution. If the result is an array, all the elements must be defined; if the result is a derived-type structure, all the components must be defined.

A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA statement, a PROGRAM statement, or another FUNCTION statement.
ENTRY statements can be included to provide multiple entry points to the subprogram.

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
```
FUNCTION

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*. (Output) 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

\[
\text{result} = \text{GETARCINFO}\ (pstart, \ pend, \ ppaint)
\]

\textit{pstart}  
(Output) Derived type xycoord. Viewport coordinates of the starting point of the arc.

\textit{pend}  
(Output) Derived type xycoord. Viewport coordinates of the end point of the arc.

\textit{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 \text{ARC} nor the \text{PIE} function has been successfully called since the last time \text{CLEARSCREEN} or \text{SETWINDOWCONFIG} was successfully called, or since a new viewport was selected.

\text{GETARCINFO} updates the \textit{pstart} and \textit{pend} xycoord derived types to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to the \text{ARC} or \text{PIE} functions. The xycoord derived type, defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory), is:

\begin{verbatim}
TYPE xycoord
\end{verbatim}
The returned value in \textit{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 \texttt{GETARCINFO}, change colors using \texttt{SETCOLORRGB}. Use the new color, along with the coordinates in \textit{ppaint}, as arguments for the \texttt{FLOODFILLRGB} function.

**Compatibility**

\texttt{STANDARD GRAPHICS QUICKWIN GRAPHICS LIB}

**See Also:** \texttt{ARC, FLOODFILLRGB, GETCOLORRGB, GRSTATUS, PIE, SETCOLORRGB}

**Example**

```fortran
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:** \texttt{USE DFLIB}

**Syntax**

```fortran
CALL GETARG (n, buffer [, status])
```

\texttt{n}
\hspace{1em} (Input) INTEGER(2). Position of the command-line argument to retrieve. The command itself is argument number 0.

\texttt{buffer}
\hspace{1em} (Output) Character*(*). Command-line argument retrieved.

\texttt{status}
\hspace{1em} (Optional; output) INTEGER(2). If specified, returns the completion status.
If there were no errors, \textit{status} returns the number of characters in the retrieved command-line argument before truncation or blank-padding. (That is, \textit{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 \textbf{NARGS}.

\textbf{GETARG} can be used with two or three arguments. If you use module DFLIB.F90 in the \texttt{\textbackslash DF98\textbackslash INCLUDE} subdirectory (by including the statement \texttt{USE DFLIB}), you can mix calls to \textbf{GETARG} with two or three arguments. If you do not use DFLIB.F90, you can use either two-argument or three-argument calls to \textbf{GETARG} but only one type of call within a subprogram.

\textbf{GETARG} returns command-line arguments as they were entered. There is no case conversion.

If the command-line argument is shorter than \textit{buffer}, \textbf{GETARG} pads \textit{buffer} on the right with blanks. If the argument is longer than \textit{buffer}, \textbf{GETARG} truncates the argument. If there is an error, \textbf{GETARG} fills \textit{buffer} with blanks.

Assume a command-line invocation of ANOVA -g -c -a, and that \textit{buffer} is at least five characters long. The following \textbf{GETARG} statements return the corresponding arguments in \textit{buffer}:

<table>
<thead>
<tr>
<th>Statement</th>
<th>String returned in \textit{buffer}</th>
<th>Length returned in \textit{status}</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL GETARG (0, buffer, status)</td>
<td>ANOVA</td>
<td>5</td>
</tr>
<tr>
<td>CALL GETARG (1, buffer)</td>
<td>-g</td>
<td>undefined</td>
</tr>
<tr>
<td>CALL GETARG (2, buffer, status)</td>
<td>-c</td>
<td>2</td>
</tr>
<tr>
<td>CALL GETARG (3, buffer)</td>
<td>-a</td>
<td>undefined</td>
</tr>
<tr>
<td>CALL GETARG (4, buffer, status)</td>
<td>all blanks</td>
<td>-1</td>
</tr>
</tbody>
</table>

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \textbf{NARGS}, \textbf{IARGC}

\textbf{Example}

\begin{verbatim}
USE DFLIB
INTEGER(2) result
result = RUNQQ('prog', '-c -r')
\end{verbatim}
GETARG

! 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 REMAPPALLETERGB.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETBKCOLORRGB, SETBKCOLOR, GETCOLOR, GETTEXTCOLOR, REMAPPALLETERGB, REMAPPALLETERGB

Example
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0</td>
<td>E B B B B B B B G G G G G G R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```fortran
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

```fortran
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 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. Note that PEEKCHARQQ is only supported in console applications.

If your application is a QuickWin or Standard Graphics application, you may want to put a call to PASSDIRKEYSQQ in your program. This will enable the program to get characters that would otherwise be trapped. These extra
characters are described in PASSDIRKEYS.

Note that the GETCHARQ routine used in a console application is a different routine than the one used in a QuickWin or Standard Graphics application. The GETCHARQ used with a console application does not trap characters that are used in QuickWin for a special purpose, such as scrolling. Console applications do not need, and cannot use PASSDIRKEYS.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: PEEKCHARQQ, GETSTRQQ, INCHARQQ, MBINCHARQQ, GETC, FGETC, PASSDIRKEYSQQ

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 (*,*)
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**

```fortran
! 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**

```plaintext
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:

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>B B E B B B G G G G G G</td>
<td>R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 x86 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:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Hex value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCW$MCW_IC</td>
<td>#1000</td>
<td>Infinity control mask</td>
</tr>
<tr>
<td>FPCW$AFFINE</td>
<td>#1000</td>
<td>Affine infinity</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>FPCW$PROJECTIVE</th>
<th>#0000</th>
<th>Projective infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCW$MCW_PC</td>
<td>#0300</td>
<td>Precision control mask</td>
</tr>
<tr>
<td>FPCW$64</td>
<td>#0300</td>
<td>64-bit precision</td>
</tr>
<tr>
<td>FPCW$53</td>
<td>#0200</td>
<td>53-bit precision</td>
</tr>
<tr>
<td>FPCW$24</td>
<td>#0000</td>
<td>24-bit precision</td>
</tr>
<tr>
<td>FPCW$MCW_RC</td>
<td>#0C00</td>
<td>Rounding control mask</td>
</tr>
<tr>
<td>FPCW$CHOP</td>
<td>#0C00</td>
<td>Truncate</td>
</tr>
<tr>
<td>FPCW$UP</td>
<td>#0800</td>
<td>Round up</td>
</tr>
<tr>
<td>FPCW$DOWN</td>
<td>#0400</td>
<td>Round down</td>
</tr>
<tr>
<td>FPCW$NEAR</td>
<td>#0000</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>FPCW$MSW_EM</td>
<td>#003F</td>
<td>Exception mask</td>
</tr>
<tr>
<td>FPCW$INVALID</td>
<td>#0001</td>
<td>Allow invalid numbers</td>
</tr>
<tr>
<td>FPCW$DENORMAL</td>
<td>#0002</td>
<td>Allow denormals (very small numbers)</td>
</tr>
<tr>
<td>FPCW$ZERODIVIDE</td>
<td>#0004</td>
<td>Allow divide by zero</td>
</tr>
<tr>
<td>FPCW$OVERFLOW</td>
<td>#0008</td>
<td>Allow overflow</td>
</tr>
<tr>
<td>FPCW$UNDERFLOW</td>
<td>#0010</td>
<td>Allow underflow</td>
</tr>
<tr>
<td>FPCW$INEXACT</td>
<td>#0020</td>
<td>Allow inexact precision</td>
</tr>
</tbody>
</table>
cause run-time errors, but you can capture the interrupts with \texttt{SIGNALQQ} and branch to your own error-handling routines.

You can use \texttt{GETCONTROLFPQQ} to retrieve the current control word and \texttt{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 \textit{The Floating-Point Environment} in the \textit{Programmer's Guide}.

\textbf{Compatibility}

\texttt{CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB}

\textbf{See Also:} \texttt{SETCONTROLFPQQ, GETSTATUSFPQQ, SIGNALQQ, MATHERRQQ, CLEARSTATUSFPQQ}

\textbf{Example}

\begin{verbatim}
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
\end{verbatim}

\textbf{GETCWD}

\textbf{Portability Function:} Retrieves the path of the current working directory.

\textbf{Module:} USE DFPORT

\textbf{Syntax}

\begin{verbatim}
result = GETCWD (dirname)
\end{verbatim}

\textit{dirname}  
(Output) Character (*(*). Name of the current working directory path, including drive letter.

\textbf{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**

```fortran
! 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**
**GETDAT**

**See Also:** GETTIM, SETDAT, SETTIM, DATE, FDATE, IDATE, JDATE

**Example**

```plaintext
! 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**

```plaintext
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, CHANGEDIROQ, GETDRIVESIZEQQ, GETDRIVESQQ, GETLASTERRORQQ, SPLITPATHQQ

**Example**

```fortran
!  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: ', 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**

```fortran
result = GETDRIVESIZEQQ (drive, total, avail)
```
drive
(Input) Character*(*) String containing the letter of the drive to get information about.

total
(Output) INTEGER(4) or INTEGER(4), DIMENSION(2) or INTEGER(8) (Alpha only). Total number of bytes on the drive.

avail
(Output) INTEGER(4) or INTEGER(4), DIMENSION(2) or INTEGER(8) (Alpha only). Number of bytes of available space on the drive.

Results:
The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The data types and dimension (if any) specified for the total and avail arguments must be the same. Specifying an array of two INTEGER(4) elements, or an INTEGER(8) argument, allows drive sizes larger than 2147483647 to be returned.

If an array of two INTEGER(4) elements is specified, the least-significant 32 bits are returned in the first element, the most-significant 32 bits in the second element. If an INTEGER(4) scalar is specified, the least-significant 32 bits are returned.

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, 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 'A  CD ' 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
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**

```fortran
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**

```fortran
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

GETEXCEPTIONPTRSQQ

Run-Time Function: Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.
Module: USE DFLIB

Syntax

\[
\text{result} = \text{GETEXCEPTIONPTRSQQ}() 
\]

Results:

The result type is INTEGER(4). The return value is the address of a data structure whose members are pointers to exception information captured by the C runtime at the time of an exception. This result value can then be used as the EPTR argument to routine TRACEBACKQQ to generate a stack trace from a user-defined handler or to inspect the exception context record directly.

Calling GETEXCEPTIONPTRSQQ is only valid within a user-defined handler that was established with SIGNALQQ or a direct call to the C rtl signal( ) function.

For a full description of exceptions and error handling, see The Floating Point Environment in the Programmer's Guide.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

See Also: TRACEBACKQQ, GETSTATUSFPQQ, CLEARSTATUSFPQQ, SETCONTROLFPQQ, GETCONTROLFPQQ, SIGNALQQ, MATHERRQQ

Example

```
PROGRAM SIGTEST
USE DFLIB
...
R3 = 0.0E0
STS = SIGNALQQ(MY_HANDLER)
! Cause a divide by zero exception
R1 = 3.0E0/R3
...
END

INTEGER(4) FUNCTION MY_HANDLER(SIGNUM,EXCNUM)
USE DFLIB
...
EPTRS = GETEXCEPTIONPTRSQQ()
...
CALL TRACEBACKQQ("Application SIGFPE error!",USER_EXIT_CODE=-1,EPTR=EPTRS)
...
MY_HANDLER = 1
END
```

A complete working example can be found in the online samples.
**GETEXITQQ**

**QuickWin Function:** Gets the setting for a QuickWin application's exit behavior.

**Module:** USE DFLIB

**Syntax**

\[
\text{result} = \text{GETEXITQQ}\left( \right)
\]

**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 you choose Yes, the application closes the window and terminates. If you choose No, the dialog box disappears and you can manipulate the window 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.

The default for both QuickWin and Console Graphics applications is **QWIN$EXITPROMPT**.

**Compatibility**

STANDARD GRAPHICS QUICKWIN.EXE LIB

**See Also:** SETEXITQQ, Using QuickWin

**Example**

```plaintext
! 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:

<table>
<thead>
<tr>
<th>If this bit flag is set</th>
<th>The file is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE$ARCHIVE</td>
<td>Marked as having been copied to a backup device.</td>
</tr>
<tr>
<td>FILE$DIR</td>
<td>A subdirectory of the current directory. Each MS-DOS directory contains two special files, &quot;.&quot; and &quot;..&quot;. 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.</td>
</tr>
<tr>
<td>FILE$HIDDEN</td>
<td>Hidden. It does not appear in the directory list you request from the command line, the Microsoft visual development environment browser, or File Manager.</td>
</tr>
<tr>
<td>FILE$READONLY</td>
<td>Write-protected. You can read the file, but you cannot make changes to it.</td>
</tr>
<tr>
<td>FILE$SYSTEM</td>
<td>Used by the operating system.</td>
</tr>
<tr>
<td>FILE$VOLUME</td>
<td>A logical volume, or partition, on a physical disk drive. This type of file appears only in the root directory of a physical device.</td>
</tr>
</tbody>
</table>

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$NOENT: The file or path specified was not found.
- ERR$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.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

**See Also:** [SETFILEACCESSQQ](#), [SETFILETIMEQQ](#), [UNPACKTIMEQQ](#)

**Example**

```plaintext
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)
GETFONINFO

Graphics Function: Gets the current font characteristics.

Module: USE DFLIB

Syntax

result = GETFONINFO (font)

font
(Output) Derived type 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 GETFONINFO.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: GETGTEXTTEXTENT, 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

GETGRID

Portability Function: Retrieves the group ID of the user of a process.

Module: USE DFPORT

Syntax

result = GETGRID ( )

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 = GETGRID( )

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**, **GETGTEXTEXTENT**

**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

**GETGTEXTEXTROTATION**

**Graphics Function:** Returns the current orientation of the font text output by **OUTGTEXT**.

**Module:** USE DFLIB

**Syntax**

\[
\text{result = GETGTEXTEXTROTATION ( )}
\]

**Results:**

The result type is INTEGER(4). It is the 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.
GETGTEXTROTATION

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**

```plaintext
result = GETHWNDQQ (unit)
```

- **unit** (Input) INTEGER(4). Window unit number. If **unit** is set to QWINFRAMEWINDOW (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**

```plaintext
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:

<table>
<thead>
<tr>
<th>This run-time routine</th>
<th>Produces these errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNQQ</td>
<td>ERR$NOMEM, ERR$2BIG, ERR$INVAL, ERR$NOENT, ERR$NOEXEC</td>
</tr>
<tr>
<td>SYSTEMQQ</td>
<td>ERR$NOMEM, ERR$2BIG, ERR$NOENT, ERR$NOEXEC</td>
</tr>
<tr>
<td>GETDRIVESIZEQQ</td>
<td>ERR$INVAL, ERR$NOENT</td>
</tr>
<tr>
<td>Function</td>
<td>Error Codes</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>GETDRIVESQQ</td>
<td>no error</td>
</tr>
<tr>
<td>GETDRIVEDIRQQ</td>
<td>ERR$NOMEM, ERR$RANGE</td>
</tr>
<tr>
<td>CHANGEDRIVEQQ</td>
<td>ERR$INVAL, ERR$NOENT</td>
</tr>
<tr>
<td>CHANGEDIRQQ</td>
<td>ERR$NOMEM, ERR$NOENT</td>
</tr>
<tr>
<td>MAKEDIRQQ</td>
<td>ERR$NOMEM, ERR$ACCES, ERR$EXIST, ERR$NOENT</td>
</tr>
<tr>
<td>DELDIRQQ</td>
<td>ERR$NOMEM, ERR$ACCES, ERR$NOENT</td>
</tr>
<tr>
<td>FULLPATHQQ</td>
<td>ERR$NOMEM, ERR$INVAL</td>
</tr>
<tr>
<td>SPLITPATHQQ</td>
<td>ERR$NOMEM, ERR$INVAL</td>
</tr>
<tr>
<td>GETFILEINFOQQ</td>
<td>ERR$NOMEM, ERR$NOENT, ERR$INVAL</td>
</tr>
<tr>
<td>SETFILETIMEQQ</td>
<td>ERR$NOMEM, ERR$ACCES, ERR$INVAL, ERR$MFILE, ERR$NOENT</td>
</tr>
<tr>
<td>SETFILEACCESSQQ</td>
<td>ERR$NOMEM, ERR$INVAL, ERR$ACCES</td>
</tr>
<tr>
<td>DELFILESQQ</td>
<td>ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$INVAL</td>
</tr>
<tr>
<td>RENAMEFILEQQ</td>
<td>ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$XDEV</td>
</tr>
<tr>
<td>FINDFILEQQ</td>
<td>ERR$NOMEM, ERR$NOENT</td>
</tr>
<tr>
<td>PACKTIMEQQ</td>
<td>no error</td>
</tr>
<tr>
<td>UNPACKTIMEQQ</td>
<td>no error</td>
</tr>
<tr>
<td>COMMITQQ</td>
<td>ERR$BADF</td>
</tr>
<tr>
<td>GETCHARQQ</td>
<td>no error</td>
</tr>
<tr>
<td>PEEKCHARQQ</td>
<td>no error</td>
</tr>
<tr>
<td>GETSTRQQ</td>
<td>no error</td>
</tr>
<tr>
<td>GETLASTERRORQQ</td>
<td>no error</td>
</tr>
<tr>
<td>SETERRORMODEQQ</td>
<td>no error</td>
</tr>
<tr>
<td>GETENVQQ</td>
<td>ERR$NOMEM, ERR$NOENT</td>
</tr>
<tr>
<td>SETENVQQ</td>
<td>ERR$NOMEM, ERR$INVAL</td>
</tr>
<tr>
<td>SLEEPQQ</td>
<td>no error</td>
</tr>
</tbody>
</table>
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,
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:

```fortran
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

```fortran
! 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

\[
\begin{align*}
\text{result} &= \text{GETPIXELRGB} (x, y) \\
\text{result} &= \text{GETPIXELRGB}_W (wx, wy)
\end{align*}
\]

\(x, y\)
(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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 B B B E B B B G G G G G G R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Larger numbers correspond to stronger color intensity with binary 111111 (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.

GETPIXELSRGB 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>B B B B B B B</td>
<td>G G G G G G G G G G G G G</td>
<td>R R R R R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

```fortran
! 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 x86 processors.

**Module:** USE DFLIB

**Syntax**

```fortran
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 occurrence of the corresponding exception. The following table lists the status flags and their values:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Hex value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPSW$MSW_EM</td>
<td>#003F</td>
<td>Status Mask (set all flags to 1)</td>
</tr>
<tr>
<td>FPSW$INVALID</td>
<td>#0001</td>
<td>An invalid result occurred</td>
</tr>
<tr>
<td>FPSW$DENORMAL</td>
<td>#0002</td>
<td>A denormal (very small number) occurred</td>
</tr>
<tr>
<td>FPSW$ZERODIVIDE</td>
<td>#0004</td>
<td>A divide by zero occurred</td>
</tr>
<tr>
<td>FPSW$OVERFLOW</td>
<td>#0008</td>
<td>An overflow occurred</td>
</tr>
<tr>
<td>FPSW$UNDERFLOW</td>
<td>#0010</td>
<td>An underflow occurred</td>
</tr>
<tr>
<td>FPSW$INEXACT</td>
<td>#0020</td>
<td>Inexact precision occurred</td>
</tr>
</tbody>
</table>

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, CLEARSTATUSFPQQ

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 you press Return or Enter.

Using GETSTRQQ rather than READ lets you 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 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 GETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>G</td>
</tr>
</tbody>
</table>

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 (\texttt{SETTEXTCOLOR}, \texttt{SETBKCOLOR}, and \texttt{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.

\textbf{Compatibility}

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

\textbf{See Also:} \texttt{SETTEXTCOLORRGB}, \texttt{GETBKCOLORRGB}, \texttt{GETCOLORRGB}, \texttt{GETTEXTCOLOR}

\textbf{Example}

\begin{verbatim}
! 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
\end{verbatim}

\textbf{GETTEXTPosITION}

\textbf{Graphics Subroutine:} Returns the current text position.

\textbf{Module: USE DFLIB}

\textbf{Syntax}

\begin{verbatim}
CALL GETTEXTPosition ( t )
\end{verbatim}

\begin{verbatim}
t (Output) Derived type rccord. Current text position. The derived type rccord is defined in DFLIB.F90 (in the \DF98\INCLUDE subdirectory) as follows:
\end{verbatim}

\begin{verbatim}
TYPE rccord
\end{verbatim}
GETTEXTPOSITION

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
The result type is INTEGER(4). The result is the unit number corresponding to the specified Windows handle. It returns -1 if \textit{whandle} does not exist.

This routine is the inverse of \texttt{GETHWNDQQ}.

**Compatibility**

\texttt{QUICKWIN GRAPHICS LIB}

**See Also:** \texttt{GETHWNDQQ}, \texttt{Using QuickWin}

---

**GETVIEWCOORD, GETVIEWCOORD\_W**

**Graphics Subroutine:** Translates physical coordinates or window coordinates to viewport coordinates.

**Module:** \texttt{USE DFLIB}

**Syntax**

```fortran
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 \texttt{xycoord}. Viewport coordinates. The \texttt{xycoord} derived type is defined in \texttt{DFLIB.F90} (in the \texttt{DF98\INCLUDE} subdirectory) as follows:

```fortran
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 \texttt{xycoord}. Viewport coordinates. The \texttt{xycoord} derived type is defined in \texttt{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
      ! specifying extended attributes,
      ! in which case extendfontsize
      ! sets the font size.
    CHARACTER(80) title      ! The window title.
    INTEGER(2) bitsperpixel  ! The number of bits per pixel.
                          ! The next three parameters provide extended font
                          ! attributes.
```

CHARACTER(32) extendfontname  ! The name of the desired font.
INTEGER(4) extendfontsize     ! Takes the same values as
       ! fontsize, when fontsize is
       ! set to QWIN$EXTENDFONT.
INTEGER(4) extendfontattributes   ! Font attributes
       ! such as bold and italic.
END TYPE windowconfig

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

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)
ELSE IF(.NOT. status ) THEN  ! if setwindowconfig error
    status = SETWINDOWCONFIG(wc)   ! reset
    status = GETWINDOWCONFIG(wc)
! 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**

```fortran
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,
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 \texttt{SETSIZ\_EQ} to take this into account.

The client area is the area actually available to place child windows.

\textbf{Compatibility}

\texttt{QUICKWIN GRAPHICS LIB}

\textbf{See Also:} \texttt{GET\_WINDOW\_CONFIG}, \texttt{SETSIZ\_EQ}, \texttt{Using QuickWin}

\textbf{GMTIME}

\textbf{Portability Subroutine:} Returns the Greenwich mean time in an array of time elements.

\textbf{Module: \texttt{USE DFPORT}}

\textbf{Syntax}

\begin{verbatim}
CALL GMTIME (stime, tarray)
\end{verbatim}

\textit{stime}  
\textbf{(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, January 1970.

\textit{tarray}  
\textbf{(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 \textit{tarray} are returned as follows:

\begin{tabular}{|c|c|}
\hline
Element & Value \\
\hline
\hline
tarray(1) & Seconds (0-59) \\
\hline
tarray(2) & Minutes (0-59) \\
\hline
\end{tabular}
<table>
<thead>
<tr>
<th>tarray(3)</th>
<th>Hours (0-23)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tarray(4)</td>
<td>Day of month (1-31)</td>
</tr>
<tr>
<td>tarray(5)</td>
<td>Month (0-11)</td>
</tr>
<tr>
<td>tarray(6)</td>
<td>Year number in century (0-99)</td>
</tr>
<tr>
<td>tarray(7)</td>
<td>Day of week (0-6, where 0 is Sunday)</td>
</tr>
<tr>
<td>tarray(8)</td>
<td>Day of year (0-365)</td>
</tr>
<tr>
<td>tarray(9)</td>
<td>Daylight saving flag (0 if standard time, 1 if daylight saving time)</td>
</tr>
</tbody>
</table>

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** [DATE_AND_TIME](#)

**Example**

```fortran
use dfport
integer(4) stime, timearray(9)
call gmtime (stime, timearray)
print *, timearray
```

**GOTO - Assigned**

**Statement:** Transfers control to the statement whose label was most recently assigned to a variable. This feature has been deleted in Fortran 95; it was obsolescent in Fortran 90. Compaq Fortran fully supports features deleted in Fortran 95.

**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 **GOTO** 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. It is an obsolescent feature in Fortran 95.

Syntax

GOTO (label-list) [r] 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 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GRFILEWRITEERROR</td>
<td>Error writing bitmap file</td>
</tr>
<tr>
<td>$GRFILEOPENERROR</td>
<td>Error opening bitmap file</td>
</tr>
<tr>
<td>$GRIMAGEREADERERROR</td>
<td>Error reading image</td>
</tr>
<tr>
<td>$GRBITMAPDISPLAYERROR</td>
<td>Error displaying bitmap</td>
</tr>
<tr>
<td>$GRBITMAPTOOLEARGE</td>
<td>Bitmap too large</td>
</tr>
<tr>
<td>$GRMPROPERBITMAPFORMAT</td>
<td>Improper format for bitmap file</td>
</tr>
<tr>
<td>$GRFILEREADERROR</td>
<td>Error reading file</td>
</tr>
<tr>
<td>$GRNOBITMAPFILE</td>
<td>No bitmap file</td>
</tr>
<tr>
<td>$GRINVALIDIMAGEBUFFER</td>
<td>Image buffer data inconsistent</td>
</tr>
<tr>
<td>$GRINSUFFICIENTMEMORY</td>
<td>Not enough memory to allocate buffer or to complete a fill operation</td>
</tr>
</tbody>
</table>
After a graphics call, compare the return value of `GRSTATUS` to `$GROK` to determine if an error has occurred. For example:

```fortran
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`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Possible GRSTATUS error codes</th>
<th>Possible GRSTATUS warning codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARC, ARC_W</td>
<td><code>$GRINVALIDPARAMETER</code></td>
<td><code>$GRNOOUTPUT</code></td>
</tr>
<tr>
<td>CLEARSCREEN</td>
<td><code>$GRINVALIDPARAMETER</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$GRINVALIDPARAMETER</th>
<th>One or more parameters invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GRMODENOTSUPPORTED</td>
<td>Requested video mode not supported</td>
</tr>
<tr>
<td>$GRERROR</td>
<td>Graphics error</td>
</tr>
<tr>
<td>$GROK</td>
<td>Success</td>
</tr>
<tr>
<td>$GRNOOUTPUT</td>
<td>No action taken</td>
</tr>
<tr>
<td>$GRCLIPPED</td>
<td>Output was clipped to viewport</td>
</tr>
<tr>
<td>$GRPARAMETERALTERED</td>
<td>One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order</td>
</tr>
<tr>
<td>Command</td>
<td>Errors</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ELLIPSE, ELLIPSE_W</td>
<td>$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>FLOODFILLRGB</td>
<td>$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>GETARCINFO</td>
<td></td>
</tr>
<tr>
<td>GETFILLMASK</td>
<td>$GRERROR, $GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>GETFONTINFO</td>
<td>$GRERROR</td>
</tr>
<tr>
<td>GETGTEXTEXTENT</td>
<td>$GRERROR</td>
</tr>
<tr>
<td>GETIMAGE</td>
<td>$GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>GETPIXEL</td>
<td></td>
</tr>
<tr>
<td>GETPIXELRGB</td>
<td>$GRBITMAPTOOLARGE</td>
</tr>
<tr>
<td>LINETO, LINETO_W</td>
<td></td>
</tr>
<tr>
<td>LOADIMAGE</td>
<td>$GRFILEOPENERROR, $GRNOBITMAPFILE, $GRALEREADERROR,</td>
</tr>
<tr>
<td></td>
<td>$GRIMPROPERBITMAPFORMAT, $GRBITMAPTOOLARGE, $GRIMAGEREADERROR</td>
</tr>
<tr>
<td>OUTGTEXT</td>
<td></td>
</tr>
<tr>
<td>PIE, PIE_W</td>
<td>$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>POLYGON,</td>
<td></td>
</tr>
<tr>
<td>POLYGON_W</td>
<td>$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>PUTIMAGE,</td>
<td>$GRERROR, $GRINVALIDPARAMETER,</td>
</tr>
<tr>
<td>PUTIMAGE_W</td>
<td>$GRINVALIDIMAGEBUFFER, $GRBITMAPDISPLAYERROR</td>
</tr>
<tr>
<td>RECTANGLE,</td>
<td></td>
</tr>
<tr>
<td>RECTANGLE_W</td>
<td>$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>Command</td>
<td>Error Codes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>REMAPPALETTERGB</td>
<td>$GRERROR, $GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>REMAPALLPALETTERGB</td>
<td>$GRERROR, $GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>SAVEIMAGE</td>
<td>$GRFILEOPENERROR</td>
</tr>
<tr>
<td>SCROLLTEXTWINDOW</td>
<td></td>
</tr>
<tr>
<td>SETBKCOLOR</td>
<td>$GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>SETBKCOLORRGB</td>
<td>$GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>SETCLIPRGN</td>
<td></td>
</tr>
<tr>
<td>SETCOLOR</td>
<td></td>
</tr>
<tr>
<td>SETCOLORRGB</td>
<td></td>
</tr>
<tr>
<td>SETFONT</td>
<td>$GRERROR, $GRINSUFFICIENTMEMORY</td>
</tr>
<tr>
<td>SETPIXEL, SETPIXEL_W</td>
<td></td>
</tr>
<tr>
<td>SETPIXELRGB, SETPIXELRGB_W</td>
<td></td>
</tr>
<tr>
<td>SETTEXTCOLOR</td>
<td></td>
</tr>
<tr>
<td>SETTEXTCOLORRGB</td>
<td></td>
</tr>
<tr>
<td>SETTEXTPosition</td>
<td></td>
</tr>
<tr>
<td>SETTEXTWINDOW</td>
<td></td>
</tr>
<tr>
<td>SETVIEWPORT</td>
<td></td>
</tr>
<tr>
<td>SETWINDOW</td>
<td>$GRINVALIDPARAMETER</td>
</tr>
<tr>
<td>SETWRITEMODE</td>
<td>$GRINVALIDPARAMETER</td>
</tr>
</tbody>
</table>

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

**See Also:** ARC, ELLIPSE, FLOODFILLRGB, LINETO, PIE, POLYGON, REMAPALLPALETTERGB, SETBKCOLORRGB, SETCOLORRGB, SETPIXELRGB,
SETEXTCOLOURGB, SETWINDOW, SETWRITEMODE
HOSTNAM

Portability Function: Retrieves the current host computer name.

Module: USE DFPORT

Syntax

\[
result = \text{HOSTNAM}(name)
\]

\(name\)

(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

Inquiry Intrinsic Function (Generic): Returns the largest number in the model representing the same type and kind parameters as the argument.

Syntax

\[
result = \text{HUGE}(x)
\]

\(x\)

(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 \( (1 - b^{-p})b^{e_{\text{max}}} \).

Integer parameters \( r \) and \( q \) are defined in Model for Integer Data; real parameters \( b \), \( p \), and \( e_{\text{max}} \) 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), \( \text{HUGE} (X) \) has the value \( (1 - 2^{-24}) \times 2^{128} \).

**IACHAR**

**Elemental Intrinsic Function (Generic):** Returns the position of a character in the ASCII character set, even if the processor's default character set is different. In Compaq Fortran, IACHAR is equivalent to the ICHAR function.

**Syntax**

\[
\text{result} = \text{IACHAR} (c)
\]

\( c \)  
(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 \( 0 \leq \text{IACHAR}(c) \leq 127 \).

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,
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

\[
\text{result} = \text{IAND} \ (i, j)
\]

\( i \)

(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:

\[
\begin{array}{ccc}
 i & j & \text{IAND} \ (i, j) \\
1 & 1 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>IIAND</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JIAND</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIAND ¹</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>
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. For example, IARGC returns 3 for the command-line invocation of PROG1 -g -c -a.

IARGC returns a value that is 1 less than that returned by NARGS. The command is not included in the index count returned by IARGC.

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 (Specific):** 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 (Specific):** Returns a pointer to the actual argument list for the current routine.

**Syntax**

```
result = IARGPTR ( )
```

**Results:**

The result type is `INTEGER(4)` on x86 processors; `INTEGER(8)` on Alpha processors. The actual argument list is an array of values of the same type.

On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, the argument count is not present and the first element has the address of the first argument. On OpenVMS systems, the first element in the array contains the argument count; subsequent elements contain the `INTEGER (8)` address of the actual arguments.

Formal (dummy) arguments that can be omitted must be declared `VOLATILE`.
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.

For more information, see Bit Functions.

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

\[
\text{result} = \text{IBCLR} \ (i, pos)
\]

\(i\)
(Input) Must be of type integer.

\(pos\)
Must be of type integer. It must not be negative and it must be less than \(\text{BIT\_SIZE}(i)\).
The rightmost (least significant) bit of \(i\) is in position 0.

Results:

The result type is the same as \(i\). The result has the value of the sequence of bits of \(i\), except that bit \(pos\) of \(i\) is set to zero.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIBCLR</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JIBCLR</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIBCLR (^1)</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

\(^1\) Alpha only

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: BTEST, IAND, IBCHNG, IBSET, IOR, IEOR, 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 \( \text{IBCLR} \) (\( \text{POS} = V, \text{I} = 15 \)) 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**

\[
\text{result} = \text{IBITS} \ (i, \ pos, \ len)
\]

- \( i \)  
  (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 \( \text{BIT\_SIZE}(i) \). The rightmost (least significant) bit of \( i \) is in position 0.

- \( len \)  
  (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.

For more information, see [Bit Functions](#).

The model for the interpretation of an integer value as a sequence of bits is shown in [Model for Bit Data](#).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIBITS</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JIBITS</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
</tbody>
</table>
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**

\[
\text{result} = \text{IBSET} \ (i, \ pos)
\]

\[i\]

(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\(i\). The rightmost (least significant) bit of \(i\) is in position 0.

**Results:**

The result type is the same as \(i\). The result has the value of the sequence of bits of \(i\), except that bit \(pos\) of \(i\) is set to 1.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.
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
```

**ICCHAR**

**Elemental Intrinsic Function (Generic):** Returns the position of a character in the processor's 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
processor's character set. \( c \) is in the range zero to \( n - 1 \), where \( n \) is the number of characters in the character set.

For any characters \( C \) and \( D \) (capable of representation in the processor), \( C \ .LE. D \) is true only if \( \text{ICHAR}(C) \ .LE. \text{ICHAR}(D) \) is true, and \( C \ .EQ. D \) is true only if \( \text{ICHAR}(C) \ .EQ. \text{ICHAR}(D) \) is true.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICHAR 1</td>
<td>CHARACTER</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>ICHAR 2</td>
<td>CHARACTER</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>ICHAR 3</td>
<td>CHARACTER</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

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**

\[
\text{CALL IDATE} (i, j, k)
\]

- or -

\[
\text{CALL IDATE} (iarray)
\]

\( i \)
(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**
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

General 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

    cDEC$ 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:

\[
\begin{array}{ccc}
   i & j & \text{IEOR (}i, j\text{)} \\
   \hline
   1 & 1 & 0 \\
   1 & 0 & 1 \\
   0 & 1 & 1 \\
   0 & 0 & 0 \\
\end{array}
\]

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIEOR</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JIEOR</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIEOR 1</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

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:

```plaintext
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**

```fortran
result = IERRNO()
```

**Results:**

The result type is INTEGER(4). The result value is the 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):

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPERM</td>
<td>1</td>
<td>Insufficient permission for operation</td>
</tr>
<tr>
<td>ENOENT</td>
<td>2</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>ESRCH</td>
<td>3</td>
<td>No such process</td>
</tr>
<tr>
<td>EIO</td>
<td>5</td>
<td>I/O error</td>
</tr>
<tr>
<td>E2BIG</td>
<td>7</td>
<td>Argument list too long</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>8</td>
<td>File is not executable</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>12</td>
<td>Not enough resources</td>
</tr>
<tr>
<td>EACCES</td>
<td>13</td>
<td>Permission denied</td>
</tr>
<tr>
<td>EXDEV</td>
<td>18</td>
<td>Cross-device link</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>20</td>
<td>Not a directory</td>
</tr>
<tr>
<td>EINVAL</td>
<td>22</td>
<td>Invalid argument</td>
</tr>
</tbody>
</table>

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 95 and Fortran 90.

**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:
<table>
<thead>
<tr>
<th>If the Value of ( \text{expr} ) is:</th>
<th>Control Transfers To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>Statement ( \text{label1} )</td>
</tr>
<tr>
<td>Equal to 0</td>
<td>Statement ( \text{label2} )</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>Statement ( \text{label3} )</td>
</tr>
</tbody>
</table>

**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 \( \text{THETA} \) is less than or equal to the real variable \( \text{CHI} \). Control passes to statement 100 only if \( \text{THETA} \) is greater than \( \text{CHI} \).

\[
\text{IF (THETA-CHI) 50,50,100}
\]

The following example transfers control to statement 40 if the value of the integer variable \( \text{NUMBER} \) is even. It transfers control to statement 20 if the value is odd.

\[
\text{IF (NUMBER / 2*2 - NUMBER) 20,40,20}
\]

The following statement transfers control to statement 10 for \( n < 10 \), to statement 20 for \( n = 10 \), and to statement 30 for \( n > 10 \):

\[
\text{IF (n-10) 10, 20, 30}
\]

The following statement transfers control to statement 10 if \( n \leq 10 \), and to statement 30 for \( n > 10 \):

\[
\text{IF (n-10) 10, 10, 30}
\]

**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:

```fortran
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:

```fortran
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 $\text{IF}$ statement in FORTRAN 77.)

**Syntax**

\[
\begin{align*}
\text{name:} & \quad \text{IF} \ (\text{expr}) \ \text{THEN} \\
& \quad \text{block} \\
\text{ELSE IF} & \quad (\text{expr}) \ \text{THEN} \ [\text{name}] \\
& \quad \text{block} \ [\ldots] \\
\text{ELSE} & \quad [\text{name}] \\
& \quad \text{block} \\
\text{END IF} & \quad [\text{name}]
\end{align*}
\]

*name*  
(Optional) Is the name of the $\text{IF}$ 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 $\text{IF THEN}$ statement, the same name must appear in the corresponding $\text{END IF}$ 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 $\text{ELSE}$ or $\text{END IF}$ statement is encountered.

Once a true value is found or an $\text{ELSE}$ 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 $\text{ELSE}$ 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 $\text{IF THEN}$ statement in a block $\text{IF}$ construct. For example, the following statement is invalid in the block $\text{IF}$ construct:

\[
\text{IF (e) THEN I = J}
\]

This statement is translated as the following logical $\text{IF}$ statement:
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**

<table>
<thead>
<tr>
<th>Construct</th>
<th>Flow of Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (e) THEN block END IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test e</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>Execute block</td>
</tr>
<tr>
<td>IF (e) THEN block&lt;sub&gt;1&lt;/sub&gt; ELSE block&lt;sub&gt;2&lt;/sub&gt; END IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test e</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>Execute block&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>IF (e&lt;sub&gt;1&lt;/sub&gt;) THEN block&lt;sub&gt;1&lt;/sub&gt; ELSE IF (e&lt;sub&gt;2&lt;/sub&gt;) THEN block&lt;sub&gt;2&lt;/sub&gt; END IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test e&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>Execute block&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>IF (e&lt;sub&gt;3&lt;/sub&gt;) THEN block&lt;sub&gt;1&lt;/sub&gt; ELSE IF (e&lt;sub&gt;2&lt;/sub&gt;) THEN block&lt;sub&gt;2&lt;/sub&gt; ELSE IF (e&lt;sub&gt;3&lt;/sub&gt;) THEN block&lt;sub&gt;3&lt;/sub&gt; ELSE block&lt;sub&gt;4&lt;/sub&gt; END IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test e&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>Execute block&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
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**

The following example shows the simplest form of an **IF** construct:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (expr) THEN</td>
<td>IF (ABS(ADJU) .GE. 1.0E-6) THEN</td>
</tr>
<tr>
<td>block</td>
<td>TOTERR = TOTERR + ABS(ADJU)</td>
</tr>
<tr>
<td>END IF</td>
<td>QUEST = ADJU/FNDVAL</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

This construct conditionally executes the block of statements between the **IF THEN** and the **END IF** statements.

The following shows another example:

```plaintext
! 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:

```plaintext
BLOCK_A: IF (D > 0.0) THEN
    ! Initial statement for named construct
    RADIANS = ACOS(D)
    DEGREES = ACOSD(D)
    ! These two statements
    END IF BLOCK_A
    ! form a block
    END IF
```

The following example shows an **IF** construct containing an **ELSE** statement:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (expr) THEN</td>
<td>IF (NAME .LT. 'N') THEN</td>
</tr>
<tr>
<td>block1</td>
<td>IFRONT = IFRONT + 1</td>
</tr>
<tr>
<td>ELSE</td>
<td>FRLET(IFRONT) = NAME(1:2)</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>block2</td>
<td>IBACK = IBACK + 1</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (expr) THEN</td>
<td>IF (A .GT. B) THEN</td>
</tr>
<tr>
<td>block1</td>
<td>D = B</td>
</tr>
<tr>
<td></td>
<td>F = A - B</td>
</tr>
<tr>
<td>ELSE IF (expr)</td>
<td>ELSE IF (A .GT. B/2.) THEN</td>
</tr>
<tr>
<td>block2</td>
<td>D = B/2.</td>
</tr>
<tr>
<td></td>
<td>F = A - B/2.</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

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:

```plaintext
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:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (expr) THEN</td>
<td>IF (A .GT. B) THEN</td>
</tr>
<tr>
<td>block1</td>
<td>D = B</td>
</tr>
<tr>
<td></td>
<td>F = A - B</td>
</tr>
<tr>
<td>ELSE IF (expr)</td>
<td>ELSE IF (A .GT. C) THEN</td>
</tr>
<tr>
<td>block2</td>
<td>D = C</td>
</tr>
<tr>
<td></td>
<td>F = A - C</td>
</tr>
<tr>
<td>ELSE IF (expr)</td>
<td>ELSE IF (A .GT. Z) THEN</td>
</tr>
<tr>
<td>block3</td>
<td>D = Z</td>
</tr>
<tr>
<td></td>
<td>F = A - Z</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>block4</td>
<td>D = 0.0</td>
</tr>
<tr>
<td></td>
<td>F = A</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (expr) THEN</td>
<td>IF (A .LT. 100) THEN</td>
</tr>
<tr>
<td>block1</td>
<td>INRAN = INRAN + 1</td>
</tr>
<tr>
<td>IF (expr2) THEN</td>
<td>IF (ABS(A-AVG) .LE. 5.) THEN</td>
</tr>
<tr>
<td>block1a</td>
<td>INAVG = INAVG + 1</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>block1b</td>
<td>OUTAVG = OUTAVG + 1</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>block2</td>
<td>OUTRAN = OUTRAN + 1</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

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:

```fortran
! 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
    END IF
    ! Statements here executed only if i >= 100
END IF
```

**IF Directive Construct**

**General 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**

```
cDEC$ IF (expr) -or- cDEC$ IF DEFINED (name)
```
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 Intrinsic Function (Generic):** Returns the length (in bits) of the
two's complement representation of an integer.

**Syntax**

\[ \text{result} = \text{ILEN} (i) \]
$i$ 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**

\[
\begin{align*}
\text{result} &= \text{IMAGESIZE} (x1, y1, x2, y2) \\
\text{result} &= \text{IMAGESIZE_W} (w1x1, w1y1, w2x2, w2y2)
\end{align*}
\]

$x1, y1$
(Input) INTEGER(2). Viewport coordinates for upper-left corner of image.

$x2, y2$
(Input) INTEGER(2). Viewport coordinates for lower-right corner of image.

$w1x1, w1y1$
(Input) REAL(8). Window coordinates for upper-left corner of image.

$w2x2, w2y2$
(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 ($w1x1$, $w1y1$) and ($w2x2$, $w2y2$).
**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**

```plaintext
IMPLICIT type (a[, a] ...)[, type (a[, a] ...)] ...
IMPLICIT NONE
```

**type**
Is a data type specifier (CHARACTER*(*)) is not allowed).

**a**
Is a single letter, a dollar sign ($), or a range of letters in alphabetical order. The form for a range of letters is \(a_1-a_2\), where the second letter follows the first alphabetically (for example, A-C).

The dollar sign can be used at the end of a range of letters, since **IMPLICIT** interprets the dollar sign to alphabetically follow the letter Z. For example, a range of X-$ would apply to identifiers beginning with the letters X, Y, Z, or $.

**Rules and Behavior**

The **IMPLICIT** statement assigns the specified data type (and kind parameter) to all names that have no explicit data type and begin with the specified letter or range of letters. It has no effect on the default types of intrinsic procedures.
When the data type is CHARACTER*len, *len* is the length for character type. The *len* is an unsigned integer constant or an integer initialization expression enclosed in parentheses. The range for *len* is 1 to 2**31-1 for Tru64 UNIX, Linux, and Windows NT systems on Alpha processors; 1 to 65535 for OpenVMS systems and x86 processors.

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

The **IMPLICIT NONE** statement disables all implicit typing defaults. When **IMPLICIT NONE** is used, all names in a program unit must be explicitly declared. An **IMPLICIT NONE** statement must precede any **PARAMETER** statements, and there must be no other **IMPLICIT** statements in the scoping unit.

---

**Note:** To receive diagnostic messages when variables are used but not declared, you can specify the `/warn:declarations` compiler option instead of using **IMPLICIT NONE**.

---

The following **IMPLICIT** statement represents the default typing for names when they are not explicitly typed:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

**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**

```plaintext
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.

For direction keys, **INCHARQQ** returns 0xF0 as the upper 8 bits, and the ASCII code as the lower 8 bits. To allow direction keys to be read, you must use the **PASSDIRKEYSEQQ** function. The escape characters (the upper 8 bits) are different from those of **GETCHARQQ**. Note that console applications do not need, and cannot use **PASSDIRKEYSEQQ**.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

**See Also:** Using QuickWin, **GETCHARQQ**, **READ**, **MBINCHARQQ**, **GETC**, **PASSDIRKEYSEQQ**

**Example**

```plaintext
use dflib
integer*4 res
integer*2 exchar
character*1 ch, ch1

Print *,"Type X to exit, S to scroll, D to pass Direction keys"

123 continue
exchar = incharqq()
! check for escapes
! 0xE0 0x?? is a function key
! 0xF0 0x?? is a direction key
```
ch = char(rshift(exchar,8) .and. #00FF)
ch1 = char(exchar .and. #00FF)

if (ichar(ch) .eq. 224) then
   print *, "function key = ", ichar(ch), " ", ichar(ch1), " ", ch1
   goto 123
endif

if (ichar(ch) .eq. 240) then
   print *, "direction key = ", ichar(ch), " ", ichar(ch1), " ", ch1
   goto 123
endif

print *, "other key = ", ichar(ch), " ", ichar(ch1), " ", ch1

if(ch1 .eq. 'S') then
   res = passdirkeysqq(.false.)
   print *, "Entering Scroll mode"
endif

if(ch1 .eq. 'D') then
   res = passdirkeysqq(.true.)
   print *, "Entering Direction keys mode"
endif

if(ch1 .ne. 'X') go to 123
end

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 (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux 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 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

<table>
<thead>
<tr>
<th>Main Program File</th>
<th>COMMON.FOR File</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM</td>
<td>INTEGER, PARAMETER :: M=100</td>
</tr>
<tr>
<td></td>
<td>REAL, DIMENSION(M) :: Z</td>
</tr>
<tr>
<td></td>
<td>CALL CUBE</td>
</tr>
<tr>
<td></td>
<td>DO I = 1, M</td>
</tr>
<tr>
<td></td>
<td>Z(I) = X(I) + SQRT(Y(I))</td>
</tr>
<tr>
<td></td>
<td>END DO</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
<tr>
<td></td>
<td>SUBROUTINE CUBE</td>
</tr>
<tr>
<td></td>
<td>INCLUDE 'COMMON.FOR'</td>
</tr>
<tr>
<td></td>
<td>DO I = 1, M</td>
</tr>
<tr>
<td></td>
<td>X(I) = Y(I)**3</td>
</tr>
<tr>
<td></td>
<td>END DO</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
</tbody>
</table>

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

\[ \text{result} = \text{INDEX} \left( \text{string}, \text{substring} \ [, \text{back}] \right) \]

\( \text{string} \)
(Input) Must be of type character.

\( \text{substring} \)
(Input) Must be of type character.

\( \text{back} \)
(Optional; input) Must be of type logical.

Results:

The result type is default integer.

If \( \text{back} \) does not appear (or appears with the value false), the value returned is the minimum value of \( I \) such that \( \text{string} \left( I : I + \text{LEN} \left( \text{substring} \right) - 1 \right) = \text{substring} \) (or zero if there is no such value). If \( \text{LEN} \left( \text{string} \right) < \text{LEN} \left( \text{substring} \right) \), zero is returned. If \( \text{LEN} \left( \text{substring} \right) = 0 \), 1 is returned.

If \( \text{back} \) appears with the value true, the value returned is the maximum value of \( I \) such that \( \text{string} \left( I : I + \text{LEN} \left( \text{substring} \right) - 1 \right) = \text{substring} \) (or zero if there is no such value). If \( \text{LEN}(\text{string}) < \text{LEN}(\text{substring}) \), zero is returned. If \( \text{LEN}(\text{substring}) = 0 \), \( \text{LEN}(\text{string}) + 1 \) is returned.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>CHARACTER</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

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**

```fortran
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. Do not use INITIALSETTINGS to open or size child windows.

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 if you define it, since it will be called automatically during initialization.

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, SETWSIZEQQ

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 SETACTIVE.

A window has focus when it is given the focus by FOCUS, 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 an 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:** FOCUS, 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:

```plaintext
INQUIRE (FILE=name [, ERR=label] [, IOSTAT=i-var] [,)
```
INQUIRE

DEFAULTFILE=\texttt{def}] \), \texttt{slist})

\textbf{Inquiring by Unit:}

\texttt{INQUIRE ([UNIT=}\texttt{io-unit} [, ERR=\texttt{label}] [, IOSTAT=\texttt{i-var}] \texttt{slist})}

\textbf{Inquiring by Output List:}

\texttt{INQUIRE (IOLENGTH=}\texttt{len}) \texttt{out-item-list}

\textit{name}
Is a scalar default character expression specifying the name of the file for inquiry.

\textit{label}
Is the label of the branch target statement that receives control if an error occurs.

\textit{i-var}
Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

\textit{slist}
Is one or more of the following inquiry specifiers (each specifier can appear only once):

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
ACCESS & DELIM & NAMED & READWRITE \\
\hline
ACTION & DIRECT & NEXTREC & RECL \\
\hline
BINARY & EXIST & NUMBER & RECORDTYPE \\
\hline
BLANK & FORM & OPENED & SEQUENTIAL \\
\hline
BLOCKSIZE & FORMATTED & ORGANIZATION & SHARE \\
\hline
BUFFERED & IOFOCUS & PAD & UNFORMATTED \\
\hline
CARRIAGECONTROL & MODE & POSITION & WRITE \\
\hline
CONVERT & NAME & READ & \\
\hline
\end{tabular}
\end{verbatim}

\texttt{def}
Is a scalar default character expression specifying a default file pathname string. (For more information, see the \texttt{DEFAULTFILE} specifier.

\texttt{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.

**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.

```fortran
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 (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**

```fortran
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.
- **WINSELECTTEXT**: Selects text from the current window.
- **WINSELECTGRAPHICS**: Selects graphics from the current window.
- **WINSELECTALL**: 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.
- **WINSIZETO Fit**: 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**

```fortran
! 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
```

**INSTANCE (TU*X only)**

**Compaq Fortran Parallel Compiler Directive:** Specifies the availability of named common blocks.

**Syntax**

```fortran
$PAR INSTANCE { SINGLE | PARALLEL } /[cb]/ [ , /cb/ ] ...
```

* c
  Is one of the following: C (or c), !, or * (see [Syntax Rules for Parallel Directives](#)).

* cb
  Is the name of the common block. The slashes are required.
**Rules and Behavior**

The keywords are described as follows:

- **SINGLE** - Specifies that there will be a single instance of the named common blocks. This implies that all threads share the same copy, and assignments to the constituent items in the common blocks occurring in one thread affect the values of those items in the same named common blocks in other threads.

- **PARALLEL** - This is the same as specifying the OpenMP Fortran API `THREADPRIVATE` directive.

**INSTANCE SINGLE** is the default for named common blocks.

**See Also:** Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

**INT**

**Elemental Intrinsic Function (Generic):** Converts a value to integer type.

**Syntax**

\[
\text{result} = \text{INT} (a [, kind] )
\]

- **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:
o If \( a \) is of type integer, \( \text{INT} (a) = a \).

o If \( a \) is of type real and \( |a| < 1 \), \( \text{INT} (a) \) has the value zero.

If \( a \) is of type real and \( |a| \geq 1 \), \( \text{INT} (a) \) 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 \( a \).

o If \( a \) is of type complex, \( \text{INT} (a) = a \) is the value obtained by applying the preceding rules (for a real argument) to the real part of \( a \).

<table>
<thead>
<tr>
<th>Specific Name 1</th>
<th>Argument Type 2</th>
<th>Result Type 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIFIX 3</td>
<td>REAL(4)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>IINT</td>
<td>REAL(4)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>IFIX 4, 5</td>
<td>REAL(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>JFIX</td>
<td>INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>INT 6, 7</td>
<td>REAL(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIFIX</td>
<td>REAL(4)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>KINT</td>
<td>REAL(4)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>IIDINT</td>
<td>REAL(8)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>Function</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IDINT 7, 8</td>
<td>REAL(8)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIDINT</td>
<td>REAL(8)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>IIQINT</td>
<td>REAL(16)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>IQINT 7, 9</td>
<td>REAL(16)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KIQINT</td>
<td>REAL(16)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(2)</td>
<td></td>
</tr>
<tr>
<td>COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(4)</td>
<td></td>
</tr>
<tr>
<td>COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(8)</td>
<td></td>
</tr>
<tr>
<td>INT1</td>
<td>INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>INT2</td>
<td>INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>INT4</td>
<td>INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>INT8</td>
<td>INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

1. These specific functions cannot be passed as actual arguments.
2. INTEGER(8) is only available on Alpha processors; REAL(16) and COMPLEX(16) are available on OpenVMS, Tru64 UNIX, and Linux 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.
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.

**INT_PTR_KIND**

Inquiry Intrinsic Function (Specific): Returns the INTEGER KIND that will hold an address. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

**Syntax**

\[
\text{result} = \text{INT_PTR_KIND}( )
\]

**Results:**

The result type is default integer. The result is a scalar with the value equal to the value of the kind parameter of the integer data type that can represent an address on the host platform.

On Windows NT (including Windows 2000) and Windows 9* systems, the value is 4. On OpenVMS, Tru64 UNIX, and Linux systems, the value is 8.

**Example**

```fortran
REAL A(100)
POINTER (P, A)
INTEGER (KIND=INT_PTR_KIND()) SAVE_P
P = MALLOC (400)
SAVE_P = P
```

**INTEGER**

**Statement:** Specifies the INTEGER data type.
**Syntax**

\[
\begin{align*}
\text{INTEGER} & \\
\text{INTEGER}([\text{KIND=}n) & \\
\text{INTEGER}^*n &
\end{align*}
\]

\(n\)

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.

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


**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)
```

**INTEGER Directive**

**General Compiler Directive:** Specifies the default integer kind.

**Syntax**

\[c\text{DEC$ INTEGER}$:{ 2 | 4 | 8 }\]
INTEGER Directive

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

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0</td>
<td>E B B B E B B B</td>
<td>G G G G G G</td>
<td>R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
**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:**

```plaintext
type, [att-ls,] INTENT (intent-spec) [, att-ls] :: d-arg [, d-arg] ...
```

**Statement:**

```plaintext
INTENT (intent-spec) ::] d-arg [, d-arg] ...
```

- `type` Is a data type specifier.
- `att-ls` Is an optional list of attribute specifiers.
- `intent-spec` Is one of the following specifiers:
  - **IN**: Specifies that the dummy argument will be used only to provide data to the procedure. The dummy argument must not be redefined (or become undefined) during execution of the procedure. Any associated actual argument must be an expression.
  - **OUT**: Specifies that the dummy argument will be used to pass data from the procedure back to the calling program. The dummy argument is undefined on entry and must be defined before it is referenced in the procedure. Any associated actual argument must be definable.
  - **INOUT**: Specifies that the dummy argument can both provide data to the procedure and return data to the calling program. Any associated actual argument must be definable.

- `d-arg` Is the name of a dummy argument. It cannot be a dummy procedure or dummy pointer.

**Rules and Behavior**
The **INTENT** statement can only appear in the specification part of a subprogram or interface body.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

If a function specifies a defined operator, the dummy arguments must have intent IN.

If a subroutine specifies defined assignment, the first argument must have intent OUT or INOUT, and the second argument must have intent IN.

A dummy argument with intent IN (or a subobject of such a dummy argument) must **not** appear as any of the following:

- A *DO* variable or implied-*DO* variable
- The variable of an assignment statement
- The *pointer-object* of a pointer assignment statement
- An *object* or STAT variable in an **ALLOCATE** or **DEALLOCATE** statement
- An input item in a **READ** statement
- A variable name in a **NAMELIST** statement if the namelist group name appears in a NML specifier in a **READ** statement
- An internal file unit in a **WRITE** statement
- A definable variable in an **INQUIRE** statement
- An IOSTAT or SIZE specifier in an I/O statement
- An actual argument in a reference to a procedure with an explicit interface if the associated dummy argument has intent OUT or INOUT

If an actual argument is an array section with a vector subscript, it cannot be associated with a dummy array that is defined or redefined (has intent OUT or INOUT).

**Compatibility**

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:

```fortran
SUBROUTINE TEST(I, J)
   INTEGER, INTENT(IN) :: I
   INTEGER, INTENT(OUT), DIMENSION(I) :: J
```
The following are examples of the **INTENT** statement:

```fortran
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:

```fortran
!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. It 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**

```fortran
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 (a Fortran 95 feature) if one appears in the **INTERFACE** statement; they must be identical.

The characteristics specified for the external or dummy procedure must be consistent with those specified in the procedure's definition.
An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

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 \textit{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 \textit{Unambiguous Generic Procedure References}.

- The module procedures listed in the \textbf{MODULE PROCEDURE} statement

  The module procedures must be accessible by a \textbf{USE} statement.

To make an interface block available to multiple program units (through a \textbf{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 \textit{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.

\textbf{Compatibility}

\textbf{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

The following example shows a simple procedure interface block with no generic specification:

```fortran
SUBROUTINE SUB_B (B, FB)
  REAL B
  ...
  INTERFACE
    FUNCTION FB (GN)
      REAL FB, GN
    END FUNCTION
  END INTERFACE
END SUBROUTINE SUB_B
```

The following shows another example:

```fortran
!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:**

```fortran
  type, [att-ls,] INTRINSIC [., att-ls] :: in-pro [, in-pro] ...
```

**Statement:**

```fortran
  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:

```fortran
PROGRAM EXAMPLE
  ...
  REAL(8), INTRINSIC :: DACOS
  ...
  CALL TEST(X, DACOS) ! Intrinsic function DACOS is an actual argument
```

The following example shows an INTRINSIC statement:

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL CTN</td>
<td>SUBROUTINE TRIG(X,F,Y)</td>
</tr>
<tr>
<td>INTRINSIC SIN, COS</td>
<td>Y = F(X)</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CALL TRIG(ANGLE,SIN,SINE)</th>
<th>FUNCTION CTN(X)</th>
</tr>
</thead>
</table>
Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Fortran 95/90 library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

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](#).
IOR

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 \textit{iflag} is 1, the generator is restarted and the first random value is returned. If \textit{iflag} is 0, the next random number in the sequence is returned. If \textit{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.

\textbf{IRAND} and \textbf{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: \texttt{RANDOM\_NUMBER}, \texttt{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

\textbf{Elemental Intrinsic Function (Generic):} Arithmetically shifts an integer left or right by a specified number of bits.

Syntax

\[
\text{result} = \text{ISHA} \ (i, \ shift)
\]

\textit{i}

(Input) Must be of type integer. This argument is the value to be shifted.

\textit{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

```fortran
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**

```plaintext
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**

\[ \text{result} = \text{ISHFT} \ (i, \ shift) \]

\( i \)

(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 \text{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 \( \text{shift} \) positions. If \( \text{shift} \) is positive, the shift is to the left; if \( \text{shift} \) is negative, the shift is to the right. If \( \text{shift} \) 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 \( \text{shift} \) can also be specified as **LSHIFT**. **ISHFT** with a negative \( \text{shift} \) can also be specified as **RSHIFT** with \(|\text{shift}|\).

For more information, see [Bit Functions](#).

The model for the interpretation of an integer value as a sequence of bits is shown in [Model for Bit Data](#).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IISHFT</td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>JISHFT</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>KISHFT ¹</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KISHFT ¹</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>

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:

```fortran
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**

```plaintext
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.

For more information, see **Bit Functions**.

The model for the interpretation of an integer value as a sequence of bits is shown in **Model for Bit Data**.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IISHFTC</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JISHFTC</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KISHFTC ¹</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>
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

```fortran
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

```fortran
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

```fortran
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**

```plaintext
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**

```plaintext
USE DFPORT
INTEGER(4) time_array(3)
CALL ITIME (time_array)
write(*,10) time_array
10 format (1X,I2,':',I2,':',I2,'
END
```

**IVDEP**

**General Compiler Directive:** Assists the compiler's dependence analysis. It can only be applied to iterative **DO** loops. This directive can also be specified as **INIT DEP FWD** (INITialize DEPendences ForWarD).

**Syntax**

```plaintext
cDEC$ IVDEP
```

- `c` is one of the following: C (or c), !, or *. (See Syntax Rules for General
Rules and Behavior

The **IVDEP** directive is an assertion to the compiler's optimizer about the order of memory references inside a **DO** loop.

The **IVDEP** directive tells the compiler to begin dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

The **IVDEP** directive must precede the **DO** statement for each **DO** loop it affects. No source code lines, other than the following, can be placed between the **IVDEP** directive statement and the **DO** statement:

- An **UNROLL** directive
- A **PARALLEL DO** directive (TU*X only)
- A **PDO** directive (TU*X only)
- Placeholder lines
- Comment lines
- Blank lines

The **IVDEP** directive is applied to a **DO** loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.

The **IVDEP** directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: General Compiler Directives

Example

In the following example, the **IVDEP** directive provides more information about
the dependences within the loop, which may enable loop transformations to occur:

```fortran
!DEC$ IVDEP
DO I=1, N
   A(INDARR(I)) = A(INDARR(I)) + B(I)
END DO
```

In this case, the scalar execution order follows:

1. Retrieve INDARR(I).
2. Use the result from step 1 to retrieve A(INDARR(I)).
4. Add the results from steps 2 and 3.
5. Store the results from step 4 into the location indicated by A(INDARR(I)) from step 1.

`IVDEP` directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.
**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**

```fortran
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**

```fortran
! 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**

```fortran
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**

```fortran
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:

```fortran
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
```

LASTPRIVATE (TU*X only)

Parallel Directive Clause: Provides a superset of the functionality provided by the PRIVATE clause; objects are declared PRIVATE and they are given certain values when the parallel region is exited.

Syntax

```
LASTPRIVATE (list)
```

- **list**
  Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause
semantics. In addition, once the parallel region is exited, each variable has the value provided by the sequentially last section or loop iteration.

When the LASTPRIVATE clause appears in a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS (or PSECTIONS) directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct.

Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS (or PSECTIONS) directive are undefined after the construct.

**LBOUND**

**Inquiry Intrinsic Function (Generic):** Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.

**Syntax**

\[
\text{result} = \text{LBOUND} (\text{array}, \text{dim}, \text{kind})
\]

*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 of *array*.

*kind*  
(Optional; input) Must be a scalar integer initialization expression.

**Results:**

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array*. Each element in the result corresponds to a dimension of *array*.

If *array* is an array section or an array expression that is not a whole array or array structure component, each element of the result has the value 1.
If *array* is a whole array or array structure component, **LBOUND** (*array, dim*) has a value equal to the lower bound for subscript *dim* of *array* (if *array(dim)* is nonzero). If *array(dim)* has size zero, the corresponding element of the result has the value 1.

The setting of compiler options that specify integer size can affect the result of this function.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: **UBOUND**

**Examples**

Consider the following:

```plaintext
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:

```plaintext
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 x86 processors.

Module: USE DFLIB

**Syntax**

```plaintext
CALL LCWRQQ (controlword)
```

*controlword*  
(Input) INTEGER(2). Floating-point processor control word.
**LCWRQQ** performs the same function as the run-time subroutine **SETCONTROLFPOQ** and is provided for compatibility.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**Example**

```fortran
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**

```fortran
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:

```fortran
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
```
LEN

Inquiry Intrinsic Function (Generic): Returns the length of a character expression.

Syntax

\[
\text{result} = \text{LEN} \ (\text{string} \ [, \ \text{kind}] )
\]

- **string**  
  (Input) Must be of type character; it can be scalar or array valued. (It can be an array of strings.)

- **kind**  
  (Optional; input) Must be a scalar integer initialization expression.

Results:

The result is a scalar of type integer. If \textit{kind} is present, the kind parameter of the result is that specified by \textit{kind}; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters in \textit{string} (if it is scalar) or in an element of \textit{string} (if it is array valued).

The setting of compiler options that specify integer size can affect the result of this function.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td>CHARACTER</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
<td>INTEGER(8) (^1)</td>
</tr>
</tbody>
</table>

\(^1\) 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:

```plaintext
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:

```plaintext
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:

```fortran
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, even if the processor's default collating sequence is different. In Compaq Fortran, **LGE** is equivalent to the `>=` operator.

**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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGE ¹</td>
<td>CHARACTER</td>
<td>LOGICAL(4)</td>
</tr>
</tbody>
</table>

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:

```fortran
LOGICAL L
L = LGE('ABC','ABD')       ! returns .FALSE.
L = LGE ('AB', 'AAAAAAB') ! returns .TRUE.
```

**LGT**

*Elemental Intrinsic Function (Generic):* Determines whether a string is lexically greater than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Compaq Fortran, **LGT** is equivalent to the > operator.

**Syntax**

```fortran
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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGT ¹</td>
<td>CHARACTER</td>
<td>LOGICAL(4)</td>
</tr>
</tbody>
</table>

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:

```fortran
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**

```fortran
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 \texttt{SETLINESTYLE}.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS

See Also: \texttt{GETCURRENTPOSITION}, \texttt{GETLINESTYLE}, \texttt{GRSTATUS}, \texttt{MOVETO}, \texttt{POLYGON}, \texttt{POLYLINEQQ}, \texttt{SETLINESTYLE}, \texttt{SETRITEMODE}

**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**

\begin{figure}[h]
\begin{center}
\includegraphics[width=0.5\textwidth]{figure.png}
\end{center}
\end{figure}

**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:** \texttt{USE DFLIB}

**Syntax**

\[
\texttt{result = LINETOAR (loc(fx), loc(fy), loc(tx) loc(ty), cnt)}
\]

$fx$
(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:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOLID</td>
<td>0</td>
</tr>
<tr>
<td>DASH</td>
<td>1</td>
</tr>
<tr>
<td>DOT</td>
<td>2</td>
</tr>
<tr>
<td>DASHDOT</td>
<td>3</td>
</tr>
<tr>
<td>DASHDOTDOT</td>
<td>4</td>
</tr>
<tr>
<td>NULL</td>
<td>5</td>
</tr>
</tbody>
</table>
**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS

**See Also:** [LINETO](#), [LINETOAR](#), [LOC](#), [POLYLINEQQ](#), [SETWRITEMODE](#)

**Example**

```fortran
! Build for QuickWin or Standard Graphics
USE DFLIB
integer(2) fx(3),fy(3),tx(3),ty(3),result
double precision 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, even if the processor's default collating sequence is different. In Compaq Fortran, LLE is equivalent to the <= operator.

**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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLE ¹</td>
<td>CHARACTER</td>
<td>LOGICAL(4)</td>
</tr>
</tbody>
</table>

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, even if the processor's default collating sequence is different. In Compaq Fortran, LLT is equivalent to the < operator.

**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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLT ¹</td>
<td>CHARACTER</td>
<td>LOGICAL(4)</td>
</tr>
</tbody>
</table>

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  
ninteger(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** specifies 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**

\[
\text{result} = \text{LOC} (x)
\]

\(x\)

(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 x86 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
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

\[
result = %\text{LOC} \,(a)
\]

\( a \) (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 \%\text{LOC} function produces an integer (INTEGER(4) on Windows NT [including
Windows 2000] and Windows 9* systems; INTEGER(8) on OpenVMS, Tru64
UNIX, and Linux 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 \%\text{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

\[
\text{result} = \text{LOG} \,(x)
\]

\(x\)

(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 \(-\pi < \omega \leq \pi\). 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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG(^1)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DLOG</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QLOG(^2)</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>CLOG(^1)</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
</tr>
<tr>
<td>CDLOG(^3)</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
</tr>
<tr>
<td>CQLOG(^2)</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
</tr>
</tbody>
</table>

\(^1\) The setting of compiler option \text{/real\_size} can affect ALOG and CLOG.
\(^2\) VMS and U*X
\(^3\) This function can also be specified as ZLOG.

Compatibility
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:

```fortran
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 $\log_{10}x$.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG10 1</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DLOG10</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QLOG10 2</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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:

```fortran
REAL r
r = LOG10(10.0)  ! returns 1.0
```

LOGICAL

Statement: Specifies the LOGICAL data type.

Syntax

```fortran
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

```fortran
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
```
LOGICAL Function

Elemental Intrinsic Function (Generic): Converts the logical value of the argument to a logical value with different kind parameters.

Syntax

\[
\text{result} = \text{LOGICAL}( l [, \text{kind}] )
\]

\(l\)
(Input) Must be of type logical.

\(\text{kind}\)
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is logical. If \(\text{kind}\) is present, the kind parameter is that specified by \(\text{kind}\); 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.

LONG

Portability Function: Returns an INTEGER(2) value as an INTEGER(4) type.

Module: USE DFPORT

Syntax
result = \textbf{LONG} (\textit{int2})

\textit{int2}  
\textbf{(Input)} \textbf{INTEGER}(2). Value to be converted.

\textbf{Results:}

The result type is \textbf{INTEGER}(4). The result is the value of \textit{int2} with type \textbf{INTEGER} (4). The upper 16 bits of the result are zeros and the lower 16 are equal to \textit{int2}.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \textbf{INT, KIND, Portability Library}

\textbf{LSHIFT}

\textbf{Elemental Intrinsic Function (Generic):} Shifts the bits in an integer left by a specified number of positions. For more information, see \textbf{ISHFT}.

\textbf{LSTAT}

\textbf{Portability Function:} Returns detailed information about a file.

\textbf{Module: USE DFPORT}

\textbf{Syntax}

\[
\text{result} = \textbf{LSTAT} \ (\textit{name, statb})
\]

\textit{name}  
\textbf{(Input)} \textbf{Character}(*). Name of the file to examine.

\textit{statb}  
\textbf{(Output)} \textbf{INTEGER}(4). One-dimensional array with a size of 12. See \textbf{STAT} for the possible values returned in \textit{statb}.

\textbf{Results:}

The result type is \textbf{INTEGER}(4). The result is zero if successful; otherwise, an error code (see \textbf{IERRNO}).

\textbf{LSTAT} returns detailed information about the file named in \textit{name}. In this implementation, \textbf{LSTAT} returns exactly the same information as \textbf{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 examine: 
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

```fortran
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

\[
\text{result} = \text{MAKEDIRQQ} (\text{dirname})
\]

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$ACCES - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- ERR$EXIST - The directory already exists.
- ERR$NOENT - The file or path specified 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
**MALLOC**

**Elemental Intrinsic Function (Specific):** Allocates a block of memory. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

**Syntax**

\[
\text{result} = \text{MALLOC} \,(i) \\
\]

\(i\)  
(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 Windows NT (including Windows 2000) and Windows 9* systems; INTEGER(8) on OpenVMS, Tru64 UNIX, and Linux systems. 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**

```plaintext
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**

```plaintext
UNION  
  MAP  
    CHARACTER*20 string  
  END MAP  
  MAP  
    INTEGER*2 number(10)  
  END MAP  
END UNION
```
MASTER (TU*X only)

OpenMP Parallel Compiler Directive: Specifies a block of code to be executed by the master thread of the team.

Syntax

```
c$OMP MASTER
    block
  c$OMP END MASTER
```

- `c` Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).
- `block` Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

When the MASTER directive is specified, the other threads in the team skip the enclosed block (section) of code and continue execution. There is no implied barrier, either on entry to or exit from the master section.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples

The following example forces the master thread to execute the routines OUTPUT and INPUT:

```
c$OMP PARALLEL DEFAULT(SHARED)
    CALL WORK(X)
c$OMP MASTER
    CALL OUTPUT(X)
    CALL INPUT(Y)
c$OMP END MASTER
    CALL WORK(Y)
```
MATHERRQQ (x86 only)

Run-Time Subroutine: Handles run-time math errors. This routine is only available on x86 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
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTH$E_DOMAIN</td>
<td>Argument domain error</td>
</tr>
<tr>
<td>MTH$E_OVERFLOW</td>
<td>Overflow range error</td>
</tr>
<tr>
<td>MTH$E_PLOSS</td>
<td>Partial loss of significance</td>
</tr>
<tr>
<td>MTH$E_SINGULARITY</td>
<td>Argument singularity</td>
</tr>
<tr>
<td>MTH$E_TLOSS</td>
<td>Total loss of significance</td>
</tr>
<tr>
<td>MTH$E_UNDERFLOW</td>
<td>Underflow range error</td>
</tr>
</tbody>
</table>
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 appropriate 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 \( \text{SUM} ((\text{row } i \text{ of } \text{matrix}_a) \times (\text{column } j \text{ of } \text{matrix}_b)) \). If the arguments are of logical type, element (i, j) of the result has the value \( \text{ANY} ((\text{row } i \text{ of } \text{matrix}_a) \text{ AND } (\text{column } j \text{ of } \text{matrix}_b)) \).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: TRANSPOSE, PRODUCT

Examples

A is matrix

\[
\begin{bmatrix}
2 & 3 & 4 \\
3 & 4 & 5 \\
\end{bmatrix},
\]

B is matrix

\[
\begin{bmatrix}
2 & 3 \\
3 & 4 \\
4 & 5 \\
\end{bmatrix},
\]

X is vector (1, 2), and Y is vector (1, 2, 3).

The result of MATMUL (A, B) is the matrix-matrix product AB with the value

\[
\begin{bmatrix}
29 & 38 \\
38 & 50 \\
\end{bmatrix}.
\]

The result of MATMUL (X, A) is the vector-matrix product XA with the value (8,
11, 14).

The result of MATMUL (A, Y) is the matrix-vector product AY with the value (20, 26).

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.

<table>
<thead>
<tr>
<th>Specific Name 1</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>REAL(4)</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>INTEGER(2)</th>
<th>REAL(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAX0</td>
<td>INTEGER(2)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>AIMAX0</td>
<td>INTEGER(2)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>MAX0 2</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>AMAX0 3, 4</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>KMAX0 5</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>AKMAX0 5</td>
<td>INTEGER(8)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>IMAX1</td>
<td>REAL(4)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>MAX1 4, 6, 7</td>
<td>REAL(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KMAX1 5</td>
<td>REAL(4)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>AMAX1 8</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DMAX1</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QMAX1 9</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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
The following shows another example:

```fortran
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**

```fortran
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 $e_{\text{max}}$, as defined in Model for Real Data.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** MINEXPONENT

**Examples**

```fortran
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**

```fortran
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. The ith subscript returned lies in the range 1 to ei, where ei is the extent of the ith dimension of array.
- If MAXLOC(array, MASK=mask) is specified, the elements in the array result form the subscript of the location of the element with the maximum value corresponding to the condition specified by mask.

The following rules apply if dim is specified:

- The array result has a rank that is one less than array, and shape (d1, d2,...d_dim-1, d_dim+1,...dn), where (d1, d2,...dn) 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 (s1, s2,...s_dim-1, s_dim+1,...s_n) of MAXLOC(array, dim [,mask]) is equal to MAXLOC(array (s1, s2,...s_dim-1, :, s_dim+1,...s_n) [, MASK = mask (s1, s2,...s_dim-1, :, s_dim+1,...s_n)]).

If more than one element has maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If array has size zero, or every element of mask has the value .FALSE., the value of the result is undefined.

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

\[
\begin{bmatrix}
  4 & 0 & -3 & 2 \\
  3 & 1 & -2 & 6 \\
  -1 & -4 & 5 & -5
\end{bmatrix}
\]

MAXLOC (A, MASK=A .LT. 5) has the value (1, 1) because these are the subscripts of the location of the maximum value (4) that is less than 5.

MAXLOC (A, DIM=1) has the value (1, 2, 3, 2). 1 is the subscript of the location of the maximum value (4) in column 1; 2 is the subscript of the location of the maximum value (1) in column 2; and so forth.

MAXLOC (A, DIM=2) has the value (1, 4, 3). 1 is the subscript of the location of the maximum value in row 1; 4 is the subscript of the location of the maximum value in row 2; and so forth.

The following shows another example:

```
INTEGER i, max
INTEGER i, max1(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  9  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

max1 = MAXLOC((/1, 4, 3, 4/)) ! returns 2, the first
! occurrence of maximum

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 = \texttt{MAXVAL} (\texttt{array [, dim] [, mask]})

\texttt{array}
\hspace*{1cm} (Input) Must be an array of type integer or real.

\texttt{dim}
\hspace*{1cm} (Optional; input) Must be a scalar integer expression with a value in the range 1 to \(n\), where \(n\) is the rank of \texttt{array}.

\texttt{mask}
\hspace*{1cm} (Optional; input) Must be a logical array that is conformable with \texttt{array}.

Results:

The result is an array or a scalar of the same data type as \texttt{array}.

The result is a scalar if \texttt{dim} is omitted or \texttt{array} has rank one.

The following rules apply if \texttt{dim} is omitted:

- If \texttt{MAXVAL} (\texttt{array}) is specified, the result has a value equal to the maximum value of all the elements in \texttt{array}.

- If \texttt{MAXVAL} (\texttt{array, MASK=mask}) is specified, the result has a value equal to the maximum value of the elements in \texttt{array} corresponding to the condition specified by \texttt{mask}.

The following rules apply if \texttt{dim} is specified:

- An array result has a rank that is one less than \texttt{array}, and shape \((d_1,\ldots,)\).
\(d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n\), where \((d_1, d_2, \ldots, d_n)\) is the shape of \texttt{array}.

- If \texttt{array} has rank one, \texttt{MAXVAL (array, dim [,mask])} has a value equal to that of \texttt{MAXVAL (array [,MASK = mask])}. Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{dim}-1}, s_{\text{dim}+1}, \ldots, s_n)\) of \texttt{MAXVAL (array, dim [,mask])} is equal to \texttt{MAXVAL (array (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n) [,MASK = mask (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n)])}.

If \texttt{array} has size zero or if there are no true elements in \texttt{mask}, the result (if \texttt{dim} is omitted), or each element in the result array (if \texttt{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 \texttt{array}.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** \texttt{MAXLOC, MINVAL, MINLOC}

**Examples**

The value of \texttt{MAXVAL ((/2, 3, 4/))} is 4 because that is the maximum value in the rank-one array.

\texttt{MAXVAL (B, MASK=B .LT. 0.0)} finds the maximum value of the negative elements of \texttt{B}.

\texttt{C} is the array

\[
\begin{pmatrix}
2 & 3 & 4 \\
5 & 6 & 7
\end{pmatrix}.
\]

\texttt{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.

\texttt{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:

\begin{verbatim}
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
\end{verbatim}
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**

\[
\text{result} = \text{MBCharLen} \ (\text{string})
\]

\[
\text{string} \\
\text{(Input) Character*(*). String containing the character whose length is to be determined. Can contain multibyte characters. }
\]

**Results:**

The result type is INTEGER(4). The result is the number of bytes in the first character contained in \text{string}. Returns 0 if \text{string} has no characters (is length 0).

\text{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**

\[
\text{result} = \text{MBConvertMBToUnicode} \ (\text{mbstr}, \text{unicodestr} [, \text{flags} ])
\]

\[
\text{mbstr}
\]
(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$ErrorInsufficientBuffer - The `unicodestr` argument is too small, but not zero size so that the needed number of bytes would be returned.
- NLS$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**
See Also: MBConvertUnicodeToMB

**MBConvertUnicodeToMB**

**NLS Function:** Converts a Unicode string to a multibyte-character string from the current codepage.

**Module:** USE DFNLS

**Syntax**

```fortran
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 \texttt{mbstr} (bytes are counted, not characters), or the number of bytes required to hold the output string if \texttt{mbstr} has zero length. If \texttt{mbstr} is longer than the translation, it is blank-padded. If \texttt{mbstr} is zero length, the function returns the number of bytes required to hold the translation and nothing is written to \texttt{mbstr}.

If an error occurs, one of the following negative values is returned:

- NLS$\text{ErrorInsufficientBuffer}$ - The \texttt{mbstr} argument is too small, but not zero length so that the needed number of bytes is returned.
- NLS$\text{ErrorInvalidFlags}$ - The \texttt{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**

\[
\text{result} = \text{MBCurMax}()
\]

**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, MBJMSToJIS**

**NLS Functions:** Converts 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**

\[
\text{result} = \text{MBLen} \ (\ string \ )
\]

\( string \)

(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**

\[
\text{result} = \text{MBLen\_Trim} \ (\ string \ )
\]
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 the 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*(*) Character to be searched 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**

```plaintext
result = MBPrev ( string, position )
```

*string*  
(Input) Character*(*) String to be searched 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**

```plaintext
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-sensitive 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**

\[
\text{result} = \text{MBVERIFY} \ ( \text{string}, \text{set} \ [, \ back \ ] )
\]

- **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 \textit{string} that is not in \textit{set}. If \textit{back} is .TRUE., returns the rightmost character in \textit{string} that is not in \textit{set}. If all the characters in \textit{string} are in \textit{set}, returns 0.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} VERIFY, MBINDEX, MBSCAN

\textbf{MERGE}

\textbf{Elemental Intrinsic Function (Generic):} Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.

\textbf{Syntax}

\begin{verbatim}
result = MERGE (tsource, fsource, mask)
\end{verbatim}

\textit{tsource}
(Input) Must be a scalar or array (of any data type).

\textit{fsource}
(Input) Must be a scalar or array of the same type and type parameters as \textit{tsource}.

\textit{mask}
(Input) Must be a logical array.

\textbf{Results:}

The result type is the same as \textit{tsource}. The value of \textit{mask} determines whether the result value is taken from \textit{tsource} (if \textit{mask} is true) or \textit{fsource} (if \textit{mask} is false).

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} MVBITS

\textbf{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

\[
\begin{bmatrix}
  1 & 3 & 5 \\
  2 & 4 & 6 
\end{bmatrix},
\]

FSOURCE is the array

\[
\begin{bmatrix}
  8 & 9 & 0 \\
  1 & 2 & 3 
\end{bmatrix},
\]

and MASK is the array

\[
\begin{bmatrix}
  T & T & T \\
  T & T & F 
\end{bmatrix}.
\]

MERGE (TSOURCE, FSOURCE, MASK) produces the result:

\[
\begin{bmatrix}
  8 & 3 & 5 \\
  2 & 4 & 3 
\end{bmatrix}.
\]

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**

**General Compiler Directive:** Specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.

**Syntax**

```
cDEC$ 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 properties 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 properties 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**

```quickwin
! Build as QuickWin app
USE DLIB
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**

```plaintext
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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN0 2</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>AMIN0 3, 4</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>KMIN0 5</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>AKMIN0 5</td>
<td>INTEGER(8)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>IMIN1</td>
<td>REAL(4)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>MIN1 4, 6, 7</td>
<td>REAL(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KMIN1 5</td>
<td>REAL(4)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>AMIN1 8</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DMIN1</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QMIN1 9</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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.
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 $e_{\text{min}}$, 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:

```fortran
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**

```fortran
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. The i\(_{th}\) subscript returned lies in the range 1 to \(e_i\), where \(e_i\) is the extent of the \(i\)th dimension of array.
- If `MINLOC(array, MASK=mask)` is specified, the elements in the array result form the subscript of the location of the element with the minimum value corresponding to the condition specified by mask.

The following rules apply if `dim` is specified:
o The array result has a rank that is one less than \texttt{array}, and shape \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) is the shape of \texttt{array}.

o If \texttt{array} has rank one, \texttt{MINLOC}(\texttt{array}, \texttt{dim} [,\texttt{mask}]) has a value equal to that of \texttt{MINLOC}(\texttt{array} [,\texttt{MASK} = \texttt{mask}]). Otherwise, the value of element \((s_1, s_2, \ldots, s_{\text{dim}-1}, s_{\text{dim}+1}, \ldots, s_n)\) of \texttt{MINLOC}(\texttt{array}, \texttt{dim} [,\texttt{mask}]) is equal to \texttt{MINLOC}(\texttt{array} (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n) [,\texttt{MASK} = \texttt{mask} (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, 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 \texttt{array} has size zero, or every element of \texttt{mask} has the value .FALSE., the value of the result is undefined.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** \texttt{MAXLOC}, \texttt{MINVAL}, \texttt{MAXVAL}

**Examples**

The value of \texttt{MINLOC} (\((/3, 1, 4, 1/))\) is (2), which is the subscript of the location of the first occurrence of the minimum value in the rank-one array.

A is the array

\[
\begin{bmatrix}
4 & 0 & -3 & 2 \\
3 & 1 & -2 & 6 \\
-1 & -4 & 5 & -5
\end{bmatrix}.
\]

\texttt{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.

\texttt{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.

\texttt{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 
! 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
min1 = 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 a scalar if \( \text{dim} \) is omitted or \( \text{array} \) has rank one.

The following rules apply if \( \text{dim} \) is omitted:

- If \( \text{MINVAL}(\text{array}) \) is specified, the result has a value equal to the minimum value of all the elements in \( \text{array} \).

- If \( \text{MINVAL}(\text{array}, \text{MASK}=\text{mask}) \) is specified, the result has a value equal to the minimum value of the elements in \( \text{array} \) corresponding to the condition specified by \( \text{mask} \).

The following rules apply if \( \text{dim} \) is specified:

- An array result has a rank that is one less than \( \text{array} \), and shape \( (d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n) \), where \( (d_1, d_2, \ldots, d_n) \) is the shape of \( \text{array} \).

- If \( \text{array} \) has rank one, \( \text{MINVAL}(\text{array}, \text{dim}, [\text{,mask}]) \) has a value equal to that of \( \text{MINVAL}(\text{array}, [\text{,MASK} = \text{mask}]) \). Otherwise, the value of element \( (s_1, s_2, \ldots, s_{\text{dim}-1}, s_{\text{dim}+1}, \ldots, s_n) \) of \( \text{MINVAL}(\text{array}, \text{dim}, [\text{,mask}]) \) is equal to \( \text{MINVAL}(\text{array} (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n) [\text{,MASK} = \text{mask} (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n)]) \).

If \( \text{array} \) has size zero or if there are no true elements in \( \text{mask} \), the result (if \( \text{dim} \) is omitted), or each element in the result array (if \( \text{dim} \) 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 \( \text{array} \).

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** MINMAX, MINLOC, MAXLOC

**Examples**

The value of \( \text{MINVAL} ((/2, 3, 4/)) \) is 2 because that is the minimum value in the rank-one array.

The value of \( \text{MINVAL} (\text{B}, \text{MASK} = \text{B} \cdot \text{GT.} \ 0.0) \) finds the minimum value of the positive elements of \( \text{B} \).

\( \text{C} \) is the array

\[
\begin{bmatrix}
2 & 3 & 4 \\
5 & 6 & 7
\end{bmatrix}
\]
MINVAL (C, DIM=1) has the value (2, 3, 4). 2 is the minimum value in column 1; 3 is the minimum value in column 2; and so forth.

MINVAL (C, DIM=2) has the value (2, 5). 2 is the minimum value in row 1 and 5 is the minimum value in row 2.

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 \( a - \text{INT}(a / p) * p \). If *p* is equal to zero, the result is undefined.
**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:

```plaintext
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
MODIFYMENUFLAGSQQ

! 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 (menuID, itemID, routine)
```

*menuID*
(Input) INTEGER(4). Identifies the menu that contains the item whose
callback routine is to 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.
- **WINSELECTTEXT**: Selects text from the current window.
- **WINSELECTGRAPHICS**: Selects graphics from the current window.
- **WINSELECTALL**: 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.
- **WIN CASCADE**: Cascades active windows.
- **WIN TILE**: Tiles active windows.
- **WIN ARRANGE**: Arranges icons.
- **WIN STATUS**: Enables a status bar.
- **WIN INDEX**: Displays the index for QuickWin Help.
- **WIN USING**: Displays information on how to use Help.
- **WIN ABOUT**: 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,
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 used by one or more program units.

Syntax

```plaintext
MODULE name
    [specification-part]
    [CONTAINS
        module-subprogram
        [module-subprogram] ...]
END [MODULE [name]]
```

name
Is the name of the module.

specification-part
Is one or more specification statements, except for the following:

- ENTRY
- FORMAT
**AUTOMATIC** (or its equivalent attribute)

**INTENT** (or its equivalent attribute)

**OPTIONAL** (or its equivalent attribute)

**Statement functions**

An automatic object must not appear in a specification statement.

*module-subprogram*

Is a function or subroutine subprogram that defines the module procedure. A function must end with **END FUNCTION** and a subroutine must end with **END SUBROUTINE**.

A module subprogram can contain internal procedures.

**Rules and Behavior**

If a name follows the **END** statement, it must be the same as the name specified in the **MODULE** statement.

The module name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A module is host to any module procedures it contains, and entities in the module are accessible to the module procedures through host association.

A module must not reference itself (either directly or indirectly).

You can use the PRIVATE attribute to restrict access to procedures or variables within a module.

Although **ENTRY** statements, **FORMAT** statements, and statement functions are not allowed in the specification part of a module, they are allowed in the specification part of a module subprogram.

Any executable statements in a module can only be specified in a module subprogram.

A module can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

**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:

```fortran
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:

```fortran
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:

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

The following example shows a module containing an interface block:

```fortran
MODULE ARRAY_CALCULATOR
    INTERFACE
        FUNCTION CALC_AVERAGE(D)
            REAL :: CALC_AVERAGE
            REAL, INTENT(IN) :: D(:)
        END FUNCTION
    END INTERFACE
END MODULE
```

The following example shows a derived-type definition that is public with components that are private:

```fortran
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:

```fortran
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:

```fortran
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 reals

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

\[
\text{result} = \text{MODULO} \ (a, \ p)
\]

\(a\)
(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 \(a - \text{FLOOR}(\text{REAL}(a) / \text{REAL}(p)) \times p\).

- If \(a\) is of type real and \(p\) is not equal to zero, the value of the result is \(a - \text{FLOOR}(a / p) \times p\).

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, wt)
```

\(x, y\)
(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

MP_SCHEDTYPE (TU*X only)

Compaq Fortran Parallel Compiler Directive: Sets a default run-time scheduling type. The scheduling type does not effect the semantics of the program, but may affect performance.

Syntax

c$PAR MP_SCHEDTYPE = mode
c
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

mode
Is one of the following:

- **DYNAMIC**

  Specifies that when a thread becomes available for more work, it is assigned the next chunksize of the remaining iterations. This is sometimes described as threads competing for iterations. If less than one chunksize of iterations remains, the next available thread is assigned all the remaining iterations.

- **GUIDED** (or GSS)

  Similar to DYNAMIC, except that the number of iterations assigned is relatively large at the beginning of the loop, and decreases exponentially as threads become available for more work. The number of iterations assigned is not necessarily divisible by chunksize.

  For this scheduling type, chunksize is the minimum number of iterations that can be assigned when a thread becomes available for work. When the number of iterations remaining to be assigned is less than or equal to chunksize, all the remaining iterations are assigned to the next available thread.

  In some cases, setting a chunksize greater than 1 improves execution efficiency as the loop nears termination, by reducing contention among the threads for the small number of remaining iterations.

- **INTERLEAVED** (or INTERLEAVE)

  Specifies that chunks of iterations are to be assigned to threads in a round-robin fashion.

- **RUNTIME**

  Specifies that environment variables are to be used to manage scheduling.

  Environment variable names are case-sensitive, but their values are not case-sensitive. Environment variables used are:

  - **MP_CHUNK** - Specifies a chunksize, where chunksize is an integer
constant.

- **MP_SCHEDTYPE** - Specifies one of the following modes: DYNAMIC, GSS, GUIDED, INTERLEAVE, INTERLEAVED, SIMPLE, or STATIC.

- **STATIC (or SIMPLE)**

Assigns each slave thread one contiguous group of iterations. Each thread is assigned an approximately equal number of iterations. STATIC is the default scheduling type when no other method has been specified.

For details on chunksize, see **CHUNK**.

**Rules and Behavior**

The **MP_SCHEDTYPE** directive can appear anywhere in a Compaq Fortran program. When more than one **MP_SCHEDTYPE** directive appears in the same program, the most recently encountered directive is used.

The scheduling type used for any parallel **DO** loop is determined from the following (in the order shown):

1. A scheduling type specified in the **PDO** directive for the current **DO** loop

2. A user-specified default specified in the most recent **MP_SCHEDTYPE** directive

3. If the scheduling type for the current **DO** loop is RUNTIME, a user-specified default specified in the environment variable **MP_SCHEDTYPE**

4. The compiler default of STATIC

The DYNAMIC and GUIDED scheduling types introduce a certain amount of overhead to manage the continuing assignment of iterations to threads during the execution of the loop. However, this overhead is sometimes offset by better load balancing when the average execution time of iterations is not uniform throughout the **DO** loop.

The STATIC and INTERLEAVED types assign all of the iterations to the threads in advance, with each thread receiving approximately equal numbers of iterations. One of these is usually the most efficient scheduling type when the average execution time of iterations is uniform throughout the **DO** loop.

The following form is also allowed: `c$MP_SCHEDTYPE = mode`
MULT_HIGH (Alpha only)

Elemental Intrinsic Function (Specific): Multiplies two 64-bit unsigned integers. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Syntax

```fortran
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:

```fortran
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
9007199254740992  2251799813685248  1099511627776
```

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).

For more information, see Bit Functions.

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:

<table>
<thead>
<tr>
<th>Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMVBITS</td>
<td>All arguments must be INTEGER(2)</td>
</tr>
<tr>
<td>JMVBITS</td>
<td>Arguments can be INTEGER(2) or INTEGER(4); at least one must be INTEGER(4)</td>
</tr>
</tbody>
</table>
### KMVBITS

<table>
<thead>
<tr>
<th>KMVBITS</th>
<th>Arguments can be INTEGER(2), INTEGER(4), or INTEGER(8); at least one must be INTEGER(8)</th>
</tr>
</thead>
</table>

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:

```fortran
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:

```plaintext
NAMELIST /LIST/ A, B, C
NAMELIST /LIST/ D, E
```

In the following example, two group names are defined:

```plaintext
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:

```fortran
REAL(4) r1
REAL(8) r2, result
r1 = 3.0
result = NEAREST (r1, -2.0)
WRITE(*,*) result ! writes 2.999999761581421

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

```fortran
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, \texttt{NINT}(a) has the value \texttt{INT}(a - 0.5).

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Specific Name} & \textbf{Argument Type} & \textbf{Result Type} \\
\hline
\texttt{ININT} & REAL(4) & INTEGER(2) \\
\hline
\texttt{NINT} \textsuperscript{1, 2} & REAL(4) & INTEGER(4) \\
\hline
\texttt{KNINT} \textsuperscript{3} & REAL(4) & INTEGER(8) \\
\hline
\texttt{IIDNNT} & REAL(8) & INTEGER(2) \\
\hline
\texttt{IDNINT} \textsuperscript{2, 4} & REAL(8) & INTEGER(4) \\
\hline
\texttt{KIDNNT} \textsuperscript{3} & REAL(8) & INTEGER(8) \\
\hline
\texttt{IIQNNT} \textsuperscript{5} & REAL(16) & INTEGER(2) \\
\hline
\texttt{IQNINT} \textsuperscript{2, 5} & REAL(16) & INTEGER(4) \\
\hline
\texttt{KIQNNT} \textsuperscript{5, 6} & REAL(16) & INTEGER(8) \\
\hline
\end{tabular}
\end{table}

\textsuperscript{1} Or \texttt{JNINT}.
\textsuperscript{2} The setting of compiler option /\texttt{integer\_size} can affect \texttt{NINT}, \texttt{IDNINT}, and \texttt{IQNINT}.
\textsuperscript{3} Alpha only
\textsuperscript{4} Or \texttt{JIDNNT}. For compatibility with older versions of Fortran, \texttt{IDNINT} can also be specified as a generic function.
\textsuperscript{5} VMS, U*X
\textsuperscript{6} This specific function cannot be passed as an actual argument.

\section*{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{ANINT, INT}

\section*{Examples}

\texttt{NINT} (3.879) has the value 4.

\texttt{NINT} (-2.789) has the value -3.

The following shows another example:

\begin{verbatim}
INTEGER(4) i1, i2
i1 = NINT(2.783) ! returns 3
i2 = IDNINT(-2.783D0) ! returns -3
\end{verbatim}
**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**

```plaintext
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**

```plaintext
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:

```plaintext
TYPE NLSEnumLocale
    CHARACTER*(NLS$MaxLanguageLen)  Language
    CHARACTER*(NLS$MaxCountryLen)   Country
```
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**

```fortran
result = NLSFormatCurrency ( outstr, instr [, flags ] )
```

- **outstr** (Output) Character*(*) (Output). String containing the correctly formatted currency for the current locale. If `outstr` is longer than the formatted currency, it is blank-padded.

- **instr** (Input) Character*(*) (Input). 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
NLSFormatCurrency

- 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$ErrorInsufficientBuffer - `outstr` buffer is too small
- NLS$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**

```fortran
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**

```fortran
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$ErrorInsufficientBuffer - *outstr* buffer is too small
- NLS$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$ErrorInsufficientBuffer - outstr buffer is too small
- NLS$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
NLSFormatNumber

\begin{verbatim}
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
\end{verbatim}

NLSFormatTime

**NLS Function:** Returns a correctly formatted string containing the time for the current locale.

**Module:** USE DFNLS

**Syntax**

\[
\text{result} = \text{NLSFormatTime} \left( \text{outstr} \ [, \text{intime} \ [, \text{flags} \] \right)
\]

- \textbf{outstr}  
  (Output) Character*(*) String containing the correctly formatted time for the current locale. If \textit{outstr} is longer than the formatted time, it is blank-padded.

- \textbf{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 \textit{intime}, the current system time is formatted and returned in \textit{outstr}.

- \textbf{flags}  
  (Optional; input) INTEGER(4). If specified, modifies the time conversion. If you omit \textit{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 \textit{outstr} (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficientBuffer - \textit{outstr} buffer is too small
- NLS$ErrorInvalidFlags - \textit{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; so, 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.

<table>
<thead>
<tr>
<th>NLS Locale Info Parameters Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>NLS$LI_ILANGUAGE</td>
</tr>
<tr>
<td>NLS$LI_SLANGUAGE</td>
</tr>
<tr>
<td>NLS$LI_SENGLANGUAGE</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>NLS$LI_SABBREVLANGNAME</td>
</tr>
<tr>
<td>NLS$LI_SNATIVELANGNAME</td>
</tr>
<tr>
<td>NLS$LI_ICOUNTRY</td>
</tr>
<tr>
<td>NLS$LI_SBCOUNTRY</td>
</tr>
<tr>
<td>NLS$LI_SENGCOUNTRY</td>
</tr>
<tr>
<td>NLS$LI_SABBREVCTRYNAME</td>
</tr>
<tr>
<td>NLS$LI_SNATIVECTRYNAME</td>
</tr>
<tr>
<td>NLS$LI_IDEFAULTLANGUAGE</td>
</tr>
<tr>
<td>NLS$LI_IDEFAULTCOUNTRY</td>
</tr>
<tr>
<td>NLS$LI_IDEFAULTANSICODEPAGE</td>
</tr>
<tr>
<td>NLS$LI_IDEFAULTOEMCODEPAGE</td>
</tr>
<tr>
<td>NLS$LI_SLIST</td>
</tr>
<tr>
<td>Symbol</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>NLS$LI_IMEASURE</td>
</tr>
<tr>
<td>NLS$LI_SDECIMAL</td>
</tr>
<tr>
<td>NLS$LI_STHOUSAND</td>
</tr>
<tr>
<td>NLS$LI_SGROUPING</td>
</tr>
<tr>
<td>NLS$LI_IDIGITS</td>
</tr>
<tr>
<td>NLS$LI_ILZERO</td>
</tr>
<tr>
<td>NLS$LI_INEGNUMBER</td>
</tr>
<tr>
<td>NLS$LI_SNATIVEDIGITS</td>
</tr>
<tr>
<td>NLS$LI_SCURRENCY</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>NLS$LI_SINTLSYMBOL</th>
<th>Three characters of the International monetary symbol specified in ISO 4217 &quot;Codes for the Representation of Currencies and Funds&quot;, followed by the character separating this string from the amount.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLS$LI_SMONDECIMALSEP</td>
<td>The character(s) used as monetary decimal separator. This is restricted such that it cannot be set to digits 0-9.</td>
</tr>
<tr>
<td>NLS$LI_SMONTHOUSANDSEP</td>
<td>The character(s) used as monetary separator between groups of digits left of the decimal. Cannot be set to digits 0-9.</td>
</tr>
<tr>
<td>&lt;NLS$LI_SMONGROUPING</td>
<td>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 &quot;3;0&quot;.</td>
</tr>
<tr>
<td>NLS$LI_ICURRDDIGITS</td>
<td>Number of decimal digits for the local monetary format.</td>
</tr>
<tr>
<td>NLS$LI_IINTLCURRDDIGITS</td>
<td>Number of decimal digits for the international monetary format.</td>
</tr>
<tr>
<td>NLS$LI_ICURRENCY</td>
<td>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 $</td>
</tr>
<tr>
<td>NLS$LI_INEGCURREN</td>
<td>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$-</td>
</tr>
<tr>
<td><strong>NLS$LI_SPOSITIVESIGN</strong></td>
<td>String value for the positive sign. Cannot be set to digits 0-9.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>NLS$LI_SNEGATIVESIGN</strong></td>
<td>String value for the negative sign. Cannot be set to digits 0-9.</td>
</tr>
</tbody>
</table>
| **NLS$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 |
<p>| <strong>NLS$LI_INEGSIGNPOSN</strong> | Determines the formatting index for negative values. Same values as for NLS$LI_IPOSSIGNPOSN. |
| <strong>NLS$LI_IPOSSEYPRECEDES</strong> | 1 if the monetary symbol precedes, 0 if it follows a positive amount. |
| <strong>NLS$LI_IPOSSEPBYSPACE</strong> | 1 if the monetary symbol is separated by a space from a positive amount, 0 otherwise. |
| <strong>NLS$LI_INEGSEYPRECEDES</strong> | 1 if the monetary symbol precedes, 0 if it follows a negative amount. |
| <strong>NLS$LI_INEGSEPBYSPACE</strong> | 1 if the monetary symbol is separated by a space from a negative amount, 0 otherwise. |</p>
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLS$LI_STIMEFORMAT</td>
<td>Time formatting string. See the NLS Date and Time Format section for explanations of the valid strings.</td>
</tr>
<tr>
<td>NLS$LI_STIME</td>
<td>Character(s) for the time separator. Cannot be set to digits 0-9.</td>
</tr>
<tr>
<td>NLS$LI_ITIME</td>
<td>Time format: 0 - Use 12-hour format 1 - Use 24-hour format</td>
</tr>
<tr>
<td>NLS$LI_ITLZERO</td>
<td>Determines whether to use leading zeros in time fields: 0 - Use no leading zeros 1 - Use leading zeros for hours</td>
</tr>
<tr>
<td>NLS$LI_S1159</td>
<td>String for the AM designator.</td>
</tr>
<tr>
<td>NLS$LI_S2359</td>
<td>String for the PM designator.</td>
</tr>
<tr>
<td>NLS$LI_SSHORTDATE</td>
<td>Short Date formatting string for this locale. The d, M and y should have the day, month, and year substituted, respectively. See the NLS Date and Time Format section for explanations of the valid strings.</td>
</tr>
<tr>
<td>NLS$LI_SDATE</td>
<td>Character(s) for the date separator. Cannot be set to digits 0-9.</td>
</tr>
<tr>
<td>NLS$LI_IDATE</td>
<td>Short Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day</td>
</tr>
<tr>
<td>NLS$LI_ICENTURY</td>
<td>Specifies whether to use full 4-digit century for the short date only: 0 - Two-digit year 1 - Full century</td>
</tr>
<tr>
<td>NLS$LI_IDAYLZERO</td>
<td>Specifies whether to use leading zeros in day fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros</td>
</tr>
<tr>
<td>NLS$LI_IMONLZERO</td>
<td>Specifies whether to use leading zeros in month fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NLS$LI_SLONGDATE</td>
<td>Long Date formatting string for this locale. The string returned may contain a string within single quotes ('). Any characters within single quotes should be left as is. The d, M and y should have the day, month, and year substituted, respectively.</td>
</tr>
<tr>
<td>NLS$LI_ILDATE</td>
<td>Long Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day</td>
</tr>
<tr>
<td>NLS$LI_ICALENDARTYPE</td>
<td>Specifies which type of calendar is currently being used: 1 - Gregorian (as in United States) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)</td>
</tr>
<tr>
<td>NLS$LI_IOPTIONALCALENDAR</td>
<td>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)</td>
</tr>
<tr>
<td>NLS$LI_IFIRSTDAYOFWEEK</td>
<td>Specifies which day is considered first in a week: 0 - SDAYNAME1 1 - SDAYNAME2 2 - SDAYNAME3 3 - SDAYNAME4 4 - SDAYNAME5 5 - SDAYNAME6 6 - SDAYNAME7</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Day of the month as digits without leading zeros for single-digit days</td>
</tr>
<tr>
<td>dd</td>
<td>Day of the month as digits with leading zeros for single-digit days</td>
</tr>
<tr>
<td>ddd</td>
<td>Day of the week as a three-letter abbreviation (SABBREVDAYNAME)</td>
</tr>
<tr>
<td>dddd</td>
<td>Day of the week as its full name (SDAYNAME)</td>
</tr>
</tbody>
</table>

#### 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.
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Month as digits without leading zeros for single-digit months</td>
</tr>
<tr>
<td>MM</td>
<td>Month as digits with leading zeros for single-digit months</td>
</tr>
<tr>
<td>MMM</td>
<td>Month as a three-letter abbreviation (SABBREVMONTHNAME)</td>
</tr>
<tr>
<td>MMMM</td>
<td>Month as its full name (SMONTHNAME)</td>
</tr>
</tbody>
</table>

Year

The year can be displayed in one of three formats using the letter "y". The table below shows the three variations:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Year represented by only the last digit</td>
</tr>
<tr>
<td>yy</td>
<td>Year represented by only the last two digits</td>
</tr>
<tr>
<td>yyyy</td>
<td>Year represented by the full 4 digits</td>
</tr>
</tbody>
</table>

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.


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; so 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**

```plaintext
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 (including Windows 2000) and Windows 9* support many locales, but these must be installed through the system Windows NT Control Panel/International menu or the Windows 9* 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**

```plaintext
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:

<table>
<thead>
<tr>
<th>I</th>
<th>NOT (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The model for the interpretation of an integer value as a sequence of bits is shown in [Model for Bit Data](#).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>INOT</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JNOT</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KNOT 1</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Alpha only</td>
</tr>
</tbody>
</table>

**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:

```fortran
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):** Initializes a pointer as disassociated when it is declared. This is a new intrinsic function in Fortran 95.

**Syntax**

```fortran
result = NULL ( [ mold ] )
```

*mold*

(Optional) 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:
The result is a pointer with disassociated association status.

**Examples**

Consider the following:

```fortran
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**

```fortran
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:

```fortran
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:

```fortran
! 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. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.
Syntax

result = NUMBER_OF_PROCESSORS ( [dim] )

(Optional) Has no effect on currently available configurations of Compaq 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 that has no generic function associated with it. It must not be passed as an actual argument. It is provided for compatibility with Compaq Fortran 77 for OpenVMS VAX systems.

Syntax

result = NWORKERS ( )

Results:

The result is always 1.
OBJCOMMENT


Syntax

```
cDEC$ 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):

<table>
<thead>
<tr>
<th>ACCESS</th>
<th>CONVERT</th>
<th>MODE</th>
<th>RECORDTYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>DEFAULTFILE</td>
<td>NAME</td>
<td>SHARE</td>
</tr>
<tr>
<td>ASSOCIATEVARIABLE</td>
<td>DELIM</td>
<td>ORGANIZATION</td>
<td>SHARED</td>
</tr>
</tbody>
</table>
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 **OPENs** can be performed on the same unit. If an **OPEN** statement is executed for a unit that already exists, the following occurs:

- If FILE is not specified, or FILE specifies the same file name that appeared in a previous **OPEN** statement, the current file remains connected.

  If the file names are the same, the values for the **BLANK**, **CARRIAGECONTROL**, **CONVERT**, **DELM**, **DISPOSE**, **ERR**, **IOSTAT**, and **PAD** specifiers can be changed. Other **OPEN** specifier values cannot be changed, and the file position is unaffected.

- If FILE specifies a different file name, the previous file is closed and the new file is connected to the unit.

The **ERR** and **IOSTAT** specifiers from any previously executed **OPEN** statement have no effect on any currently executing **OPEN** statement. If an error occurs, no file is opened or created.

Secondary operating system messages do not display when **IOSTAT** is specified. To display these messages, remove **IOSTAT** or use a platform-specific method.
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',
     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: ' filename
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:

```fortran
SUBROUTINE TEST(A)
  REAL, OPTIONAL, DIMENSION(-10:2) :: A
END SUBROUTINE
```

The following is an example of the **OPTIONAL** statement:

```fortran
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**

```fortran
SUBROUTINE TEST(ONE, TWO, THREE, FOUR)
  INTEGER ONE, TWO, THREE, FOUR
  OPTIONAL :: TWO
END SUBROUTINE
END INTERFACE
```

```fortran
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:

```fortran
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

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
OPTIONS

/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

OPTIONS Directive
General Compiler Directive: Affects data alignment and warnings about data alignment.

Syntax

```
cDEC$ OPTIONS option [option]
...
cDEC$ END OPTIONS
```

c Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

option
Is one (or both) of the following:

- `/WARN=[NO]ALIGNMENT`

  Controls whether warnings are issued by the compiler for data that is not naturally aligned. By default, you receive compiler messages when misaligned data is encountered (/WARN=ALIGNMENT).

- `/[NO]ALIGN[=p]`

  Controls alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

  p
  Is a specifier with one of the following forms:

  ```
  [class = ] rule
  (class = rule,...)
  ALL
  NONE
  ```

class
Is one of the following keywords:

- COMMONS: For common blocks
- RECORDS: For records
- STRUCTURES: A synonym for RECORDS

rule
Is one of the following keywords:
- **PACKED**
  Packs fields in records or data items in common blocks on arbitrary byte boundaries.

- **NATURAL**
  Naturally aligns fields in records and data items in common blocks on up to 64-bit boundaries (inconsistent with the Fortran 95/90 standard).
  This keyword causes the compiler to naturally align all data in a common block, including INTEGER(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 95/90 standard).
  This keyword only applies to common blocks; so, you can specify `cDEC$ OPTIONS /ALIGN=COMMONS=STANDARD`, but you cannot specify `cDEC$ OPTIONS /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, Compaq Fortran aligns local data items on natural boundaries. However, `EQUIVALENCE`, `COMMON`, `RECORD`, and `STRUCTURE` data declaration statements can force misaligned data. If `cDEC$ OPTIONS/WARN=NOALIGNMENT` is specified, warnings will not be issued if misaligned data is encountered.

**Note:** Misaligned data significantly increases the time it takes to execute a program. As the number of misaligned fields encountered increases, so does the time needed to complete program execution. Specifying `cDEC$ OPTIONS/ALIGN` (or the `/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).

**OR**

**Elemental Intrinsic Function (Generic):** Performs a bitwise inclusive OR on its arguments. For more information, see IOR.

Example

```
INTEGER i
i = OR(3, 10)   ! returns 11
```

**ORDERED (TU*X only)**

**OpenMP Parallel Compiler Directive:** Specifies a block of code to be executed in the order in which iterations would be executed in sequential
execution.

Syntax

\[
\begin{align*}
\text{c}\$\text{OMP ORDERED} \\
\text{block} \\
\text{c}\$\text{OMP ORDERED}
\end{align*}
\]

Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\[
\text{block}
\]

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. The DO directive to which the ordered section binds must have the ORDERED clause specified.

An iteration of a loop using a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

One thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it can be guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel.

Ordered sections that bind to different DO directives are independent of each other.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

\[
\begin{align*}
\text{c}\$\text{OMP DO ORDERED SCHEDULE(DYNAMIC)} \\
\text{DO I=LB,UB,ST} \\
\text{CALL WORK(I)}
\end{align*}
\]
SUBROUTINE WORK(K)
c$OMP ORDERED
  WRITE(*,*) K
c$OMP END ORDERED
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, GETGTEXTTEXTENT, 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))
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

\[
\text{result} = \text{PACK}(\text{array}, \text{mask} [, \text{vector}])
\]

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

\[
\begin{bmatrix}
0 & 8 & 0 \\
0 & 0 & 0 \\
7 & 0 & 0 \\
\end{bmatrix}
\]

PACK (N, MASK=N .NE. 0, VECTOR=(/1, 3, 5, 9, 11, 13/)) produces the result (7, 8, 5, 9, 11, 13).

PACK (N, MASK=N .NE. 0) produces the result (7, 8).

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 )
```

PACK Directive

**General Compiler Directive:** Specifies the memory starting addresses of derived-type items (and record structure items).

**Syntax**

```
cDEC$ 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 CDEC$ PACK:1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted. If CDEC$ PACK:4 is specified, INTEGER(1), LOGICAL(1), and all character variables begin at the next available byte, whether odd or even. INTEGER(2) and LOGICAL(2) begin on the next even byte; all other variables begin on 4-byte boundaries.

If the PACK directive is specified without a number, packing reverts to the compiler option setting (if any), or the default setting of 8.

The directive can appear anywhere in a program before the derived-type definition or record structure definition. It cannot appear inside a derived-type or record structure definition.

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:

**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**

```fortran
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
```

**PARALLEL (TU*X only)**

**OpenMP Parallel Compiler Directive:** Defines a parallel region.
Syntax

\[
\texttt{c}$\text{OMP PARALLEL} [\text{clause}[.,] \text{clause}] \ldots \]
\texttt{block}
\texttt{c}$\text{OMP END PARALLEL}
\]

\(c\)
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\(clause\)
Is one of the following:

- \texttt{COPYIN (list)}
- \texttt{DEFAULT ( PRIVATE | SHARED | NONE )}
- \texttt{FIRSTPRIVATE (list)}
- \texttt{IF (scalar_\_logical_\_expression)}

  Specifies that the enclosed code section is to be executed in parallel only if the \texttt{scalar_\_logical_\_expression} evaluates to \texttt{.TRUE.}. Otherwise, the parallel region is serialized. If this clause is not used, the region is executed as if an \texttt{IF(.TRUE.)} clause were specified.

  This clause is evaluated by the master thread before any data scope attributes take effect.

  Only a single IF clause can appear in the directive.

- \texttt{PRIVATE (list)}
- \texttt{REDUCTION( operator | intrinsic : list)}
- \texttt{SHARED (list)}

\(block\)
Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

Rules and Behavior

The \texttt{PARALLEL} and \texttt{END PARALLEL} directive pair must appear in the same routine in the executable section of the code.
The **END PARALLEL** directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution at the end of a parallel region.

When a thread encounters a parallel region, it creates a team of threads and it becomes the master of the team. The master thread is a member of the team and it has a thread number of zero within the team. The number of threads in the team can be controlled by environment variables and library calls.

Once created, the number of threads in the team remains constant for the duration of that parallel region. However, you can explicitly change the number of threads used in the next parallel region by calling the run-time library routine *omp_set_num_threads* from a serial portion of the program. This routine overrides any value you may have set using the environment variable OMP_NUM_THREADS. For details on environment variables, see your user manual.

The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads.

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always serialized and executed by a team of one thread.

**See Also:** Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only), Parallel Library Routines on Tru64 UNIX Systems

**Examples**

You can use the **PARALLEL** directive in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array X upon which to work based on the thread number:

```c
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
    IAM = OMP_GET_THREAD_NUM( )
    NP = OMP_GET_NUM_THREADS( )
    IPOINTS = NPOINTS/NP
    CALL SUBDOMAIN(X,IAM,IPOINTS)
    !$OMP END PARALLEL
```

Assuming you previously used the environment variable OMP_NUM_THREADS to set the number of threads to six, you can change the number of threads between parallel regions as follows:

```c
CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
```
PARALLEL DO (TU*X only)

OpenMP Parallel Compiler Directive: Provides an abbreviated way to specify a parallel region containing a single DO directive.

Syntax

\[
c\$OMP PARALLEL DO \{clause[\{,\} clause] \ldots \}
do-loop
\{c\$OMP END PARALLEL DO\}
\]

\textit{c}
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\textit{clause}
Can be any of the clauses accepted by the DO or PARALLEL directives.

\textit{do-loop}
Is a DO iteration (a DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

You cannot branch out of a DO loop associated with a DO directive.

Rules and Behavior

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples
In the following example, the loop iteration variable is private by default and it is not necessary to explicitly declare it. The `END PARALLEL DO` directive is optional:

```c
C$OMP PARALLEL DO
DO I=1,N
   B(I) = (A(I) + A(I-1)) / 2.0
END DO
C$OMP END PARALLEL DO
```

The following example shows how to use the REDUCTION clause in a `PARALLEL DO` directive:

```c
C$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
DO I=1,N
   CALL WORK(ALOCAL,BLOCAL)
   A = A + ALOCAL
   B = B + BLOCAL
END DO
C$OMP END PARALLEL DO
```

**PARALLEL SECTIONS (TU*X only)**

**OpenMP Parallel Compiler Directive:** Provides an abbreviated way to specify a parallel region containing a single `SECTIONS` directive. The semantics are identical to explicitly specifying a `PARALLEL` directive immediately followed by a `SECTIONS` directive.

**Syntax**

```c
C$OMP PARALLEL SECTIONS [clause[][[,] clause] ... ]
[C$OMP SECTION]
   block
[C$OMP SECTION]
   block] ...
C$OMP END PARALLEL SECTIONS
```

- `c`
  Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

- `clause`
  Can be any of the clauses accepted by the `PARALLEL` or `SECTIONS` directives.

- `block`
  Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.
The last section ends at the **END PARALLEL SECTIONS** directive.

**See Also:** Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

**Examples**

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```c
C$OMP PARALLEL SECTIONS
C$OMP SECTION
CALL XAXIS
C$OMP SECTION
CALL YAXIS
C$OMP SECTION
CALL ZAXIS
C$OMP END PARALLEL SECTIONS
```

**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:**

```c
type, [att-Is,] PARAMETER [, att-Is] :: c = expr [, c = expr ] ...
```

**Statement:**

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

*type*  
Is a data type specifier.

*att-Is*  
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:

```plaintext
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**, **PARAMETER**
Alternative syntax for the PARAMETER statement.

Examples

The following example shows a type declaration statement specifying the PARAMETER attribute:

```plaintext
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
```

The following is an example of the PARAMETER statement:

```plaintext
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:

```plaintext
! 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
```

**PASSDIRKEYSQQQ**

QuickWin Function: Determines the behavior of direction and page keys in a QuickWin application.

Module: USE DFLIB

Syntax

```plaintext
result = PASSDIRKEYSQQQ (val)
```

`val`
(Input) INTEGER(4) or LOGICAL(4).

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.

The following constants, defined in DFLIB.F90 in the \DF98\INCLUDE subdirectory), can be used as integer arguments:

- **PASS_DIR_FALSE** - Turns off any special handling of direction keys. They are not passed to the program by `GETCHARQQ`.
- **PASS_DIR_TRUE** - Turns on special handling of direction keys. That is, they are passed to the program by `GETCHARQQ`.
- **PASS_DIR_INSDEL** - INSERT and DELETE are also passed to the program by `GETCHARQQ`.
- **PASS_DIR_CNTRL** - Only needed for a QuickWin application, but harmless if used with a Standard Graphics application that already passes CTRL+C. This value allows CTRL+C to be passed to a QuickWin program by `GETCHARQQ` if the following is true: the program must have removed the File menu EXIT item by using `DELETEMENUQQ`. This value also passes direction keys and INSERT and DELETE.

### Results:

The return value indicates the previous setting of the PassDirKeys flag.

The return data type is the same as the data type of `val`; that is, either INTEGER (4) or LOGICAL(4).

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 **PASSDIRKEYS** 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`.

### Examples

```fortran
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

The following example uses an integer constant as an argument to
PASSDIRKEYSQQ:

=======================================================================
c
c dirkeys4.for
c
The following sample requires a compiler greater that CVF6.1.
c=======================================================================
c
Compile/Load Input Line for Standard Graphics Full Screen Window
f90 /libs:gwins dirkeys4.for
c
Compile/Load Input Line for QuickWin Graphics
f90 /libs:gwin dirkeys4.f90
c
Program to illustrate how to get almost every character
from the keyboard in QuickWin or Standard Graphics mode.
Comment out the deletemenu line for Standard Graphics mode.
c
If you are doing a standard graphics application,
control C will come in as a #03 without further
effort.
c
In a QuickWin application, The File menu Exit item must
be deleted, and PassDirKeysQQ called with PASS_DIR_CNTRLC
c to get control C.
c=======================================================================
use dfplib

integer(4) status

character*1 key1,key2,ch1

write(*,*) 'Initializing'

c------don't do this for a Standard Graphics application

status = deletemenuqq(1,3) ! stop QuickWin from getting control C

c-------set up to pass all keys to window including control c.

status = passdirkeysqq(PASS_DIR_CNTRLC)

c===============================================================

c     read and print characters

c===============================================================

10 key1 = getcharqq()

c-------first check for control+c

if(ichar(key1) .eq. 3) then
    write(*,*) 'Control C Received'
    write(*,*) 'Really want to quit?'
    write(*,*) 'Type Y to exit, or any other char to continue.'
    read(*,*) ch1
    if(ch1.eq."y" .or. ch1.eq."Y") goto 30
    goto 10
endif

if(ichar(key1).eq.0) then ! function key?
    key2 = getcharqq()
    write(*,15) ichar(key1),ichar(key2),key2
else
    if(ichar(key1).eq.224) then ! direction key?
        key2 = getcharqq()
        write(*,16) ichar(key1),ichar(key2),key2
    else
        write(*,20) key1,ichar(key1) ! normal key
    endif
endif

format(1x,2i12,1x,a1,' function key')

15 format(1x,2i12,1x,a1,' function key')

16 format(1x,2i12,1x,a1,' direction key')

20 format(1x,a1,i11)

end

go to 10

30 stop

end

**PAUSE**

**Statement:** Temporarily suspends program execution and lets you execute operating system commands during the suspension. The **PAUSE** statement is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Compaq 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 (including Windows 2000) and Windows 9* systems:
  
  `Fortran Pause - Enter command<CR> or <CR> to continue.`

- On OpenVMS systems, the system prompt

- On Tru64 UNIX and Linux systems:

  `PAUSE prompt>`

**Effect on Windows NT (including Windows 2000) and Windows 9* 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:
PAUSE

- **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 Tru64 UNIX and Linux 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:
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)
```

**PDO (TU*X only)**

**Compaq Fortran Parallel Compiler Directive:** Specifies that the iterations of the immediately following DO loop must be executed in parallel.

**Syntax**

```
c$PAR PDO [option[[,] option] ... ]
do_loop
[c$PAR END PDO [NOWAIT]]
```

c
Is one of the following: **C** (or c), **!**, or ***** (see Syntax Rules for Parallel Directives).

**option**
Is one of the following:

- **CHUNK** (chunksize)

  Adjusts the number of consecutive iterations assigned to a thread. At the end of the PDO construct, chunksize reverts to the default. The effect of CHUNK varies, depending on the scheduling type.

  A chunksize specified in a PDO directive supersedes any chunksize set with a CHUNK directive earlier in the program, and applies only for the duration of the PDO construct.

  For details on default chunksize and on the effect of specifying chunksizes for the same program in more than one context, see CHUNK (TU*X).

  BLOCKED is allowed as an alternative spelling for CHUNK.

- **FIRSTPRIVATE** (list)
The LASTLOCAL keyword is the same as the LASTPRIVATE clause.

LAST LOCAL is allowed as an alternative spelling for LASTLOCAL, even in free form source code.

[MP_SCHEDTYPE=} mode

Controls the scheduling type and allocation of work for the PDO construct.

There are several ways to specify a scheduling type. The scheduling type defaults to STATIC when no other information is available. For a description of mode and scheduling types, see MP_SCHEDTYPE (TU*X only).

At the end of the PDO construct, the scheduling type reverts to the default. The scheduling type does not affect the correctness of the program, but may affect performance.

(OBJECTED)

Specifies that iterations are assigned to threads in the same order in which iterations would be generated by a normal Compaq Fortran DO statement.

PRIVATE (list)

LOCAL is allowed as an alternative spelling for PRIVATE.

REDUCTION (var[[,] var] ...)

The REDUCTION keyword used here is different from the REDUCTION clause described previously, where both a variable and an operator type need to be specified.

In this version of the keyword, the operator is not given in the PDO directive. The compiler must be able to determine the reduction operation from the source code.

The REDUCTION keyword can be applied to a variable in a DO loop only if the variable meets the following criteria:

- It must be scalar.
- It must be assigned to exactly once in the DO loop.

- It must be read from exactly once in the DO loop and only in the right side of the assignment.

- The assignment must be one of the following forms:

  
  
  \[
  x = x \text{ operator expr}
  \]

  
  \[
  x = \text{expr operator } x \text{ (except for subtraction)}
  \]

  
  \[
  x = \text{operator}(x, \text{expr})
  \]

  
  \[
  x = \text{operator}(\text{expr}, x)
  \]

  
  where \text{operator} is one of the following supported reduction operations: +, −, *, .AND., .OR., .EQV., .NEQV., \text{MAX, MIN, IAND, or IOR}.

The compiler rewrites the reduction operation by computing partial results into local variables and then combining the results into the reduction variable. The reduction variable must be SHARED in the enclosing context.

\text{do\_loop}

Is a Compaq Fortran DO construct with loop control. It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

The iterations of the DO loop are distributed across the already existing threads.

Rules and Behavior

Specifying NOWAIT is the same as specifying NOWAIT in an OpenMP Fortran API DO directive.

PDO directives are permitted only within the lexical extent of the PARALLEL and END PARALLEL directives.

For more information about rules and restrictions, see your user manual.

See Also: Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

PDONE (TU*X only)
**Compaq Fortran Parallel Compiler Directive:** Is an executable statement that terminates the current parallel DO loop.

**Syntax**

```c
$PAR PDONE
```

`c`
Is one of the following: C (or c), !, or * (see **Syntax Rules for Parallel Directives**).

**Rules and Behavior**

The **PDONE** directive terminates the current parallel DO loop early, without completing any iterations that have not yet been assigned to a thread. Any iterations that have already been assigned to a thread are completed.

**PDONE** directives are permitted only within the lexical extent of a **PDO** or **PARALLEL DO** construct.

When the scheduling type is STATIC or INTERLEAVED, a **PDONE** directive has no effect because all iterations are assigned before execution of the parallel DO loop begins.

**See Also:** [Parallel Directives for Tru64 UNIX Systems](#), [Compaq Fortran Parallel Compiler Directives](#) (TU*X only), [OpenMP Fortran API Compiler Directives](#) (TU*X only)

---

**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**

```fortran
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, GETSTRQQ, FGETC, GETC**

**Example**

```fortran
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**

```fortran
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:** **GERROR, IERRNO**

**Example**
USE DFPRT
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**

\[
\text{result} = \text{PIE} (i, x1, y1, x2, y2, x3, y3, x4, y4) \\
\text{result} = \text{PIE_W} (i, wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4)
\]

\(i\)
(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 95/90

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.

\textit{att-ls}
Is an optional list of attribute specifiers.

\textit{ptr}
Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.

\textit{d-spec}
(Optional) Is a deferred-shape specification (\ldots). 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 \texttt{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 \texttt{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 \texttt{DATA}, \texttt{EQUIVALENCE}, or \texttt{NAMELIST} statement.

Fortran 90 pointers are not the same as integer pointers. For more information, see the \texttt{POINTER - Compaq Fortran} statement.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** \texttt{ALLOCATE}, \texttt{ASSOCIATED}, \texttt{DEALLOCATE}, \texttt{NULLIFY}, \texttt{TARGET}, \texttt{Deferred-Shape Arrays}, \texttt{Pointer Assignments}, \texttt{Pointer Association}, \texttt{Pointer Arguments}, \texttt{NULL}, \texttt{Compaq Fortran POINTER statement}, \texttt{Type Declarations}, \texttt{Compatible attributes}.

**Examples**

The following example shows type declaration statements specifying the \texttt{POINTER} attribute:
The following is an example of the **POINTER** statement:

```fortran
TYPE(SYSTEM) :: TODAYS
POINTER :: TODAYS, A(:,:)
```

See also the examples **POINTER.F90** and **POINTER2.F90** in `/DF98/SAMPLES/TUTORIAL`.

The following shows another example:

```fortran
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 - Compaq 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 95/90 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 (including Windows 2000) and Windows 9* systems, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER(4)). On OpenVMS, Tru64 UNIX, and Linux systems, a pointer occupies two numeric storage units, so it is a 64-bit quantity (INTEGER(8)).
- A pointer cannot be a pointee.
- A pointer cannot appear in an **ASSIGN** statement and cannot have the following attributes:

```
ALLOCATABLE   INTRINSIC   POINTER
EXTERNAL       PARAMETER   TARGET
```

A pointer can appear in a **DATA** statement with integer literals only.
- Integers can be converted to pointers, so you can point to absolute memory locations.
- A pointer variable cannot be declared to have any other data type.
- A pointer cannot be a function return value.
- You can give values to pointers by doing the following:
  - Retrieve addresses by using the **LOC** intrinsic function (or the %LOC built-in function)
  - Allocate storage for an object by using the **MALLOC** intrinsic function
    (or by using malloc(3f) on Tru64 UNIX or Linux systems, or LIB$GET_VM on OpenVMS systems)

For example:

**Using %LOC:**

**Using MALLOC:**
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 (Compaq 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)
INTEGER j(2)

! This has the same effect as j(1) = 0, j(2) = 5
p = LOC(j)
k = 0
p = p + SIZEOF(k) ! 4 for 4-byte integer
k = 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
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

REAL(8) wx
REAL(8) wy
END TYPE wxycoord
POLYLINEQQ

Graphics Function: Draws a line between each successive x, y point in a given array.

Module: USE DFLIB

Syntax

result = POLYLINEQQ(points, cnt)

points
(Input) An array of DF_POINT objects. The derived type DF_POINT is defined in DFLIB.F90 as:

type DF_POINT
  sequence
    integer(4) x
    integer(4) y
end type DF_POINT

cnt
(Input) INTEGER(4). Number of elements in the points array. On Windows 9* systems, cnt must be less than or equal to 16383. This is a limitation of the operating system.
**Results:**

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, zero.

**POLYLINEQQ** uses the viewport-coordinate system.

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**.

The current graphics position is not used or changed as it is in the **LINETO** function.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS

**See Also:** **LINETO**, **LINETOAREX**, **SETCOLORRGB**, **SETLINESTYLE**, **SETWRITEMODE**

**Example**

```plaintext
! Build for QuickWin or Standard Graphics
USE DFLIB
TYPE(DF_POINT) points(12)
integer(4) result
type( integer(4) ) cnt, i
! load the points
do i = 1,12,2
   points(i).x =20*i
   points(i).y =10
   points(i+1).x =20*i
   points(i+1).y =60
end do
! A sawtooth pattern will appear in the upper left corner
result = POLYLINEQQ(points, 12)
end
```

**POPCNT**

**Elemental Intrinsic Function (Generic):** Returns the number of 1 bits in an integer.

**Syntax**

```plaintext
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 \texttt{B'0...0011010110'} , the value of \( \text{POPCNT}(I) \) is 5.

**POPPAR**

**Elemental Intrinsic Function (Generic):** Returns the parity of an integer.

**Syntax**

\[
\text{result} = \text{POPPAR} (i) \\
\]

\( i \)

Integer.

Results:

The result type is the same as \( i \). If there are an odd number of 1 bits in the binary representation of the integer \( I \), the result value is 1. If there are an even number, the result value is zero.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of \( I \) is \texttt{B'0...00011010110'} , the value of \( \text{POPPAR}(I) \) 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**

\[
\text{result} = \text{PRECISION} (x) \\
\]
PRECISION

(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 \texttt{INT} \((\textsc{DIGITS}(x) - 1) \times \log_{10}(\textsc{RADIX}(x)))\). If \textsc{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 \texttt{REAL(4)} value, \texttt{PRECISION(X)} has the value 6. The value 6 is derived from \(\text{INT} \((24-1) \times \log_{10}(2.)\)) = \text{INT} (6.92...)\). For more information on the model for \texttt{REAL(4)}, see \texttt{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

\[
\text{result} = \texttt{PRESENT} \ (a)
\]

\(a\)

(Input) Must be an optional argument of the current procedure.

Results:

The result type is default logical scalar. The result is \texttt{.TRUE.} if \(a\) is present; otherwise, the result is \texttt{.FALSE.}.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \texttt{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
700     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:
**Statement:**

\[
\text{PRIVATE } [\text{::} \text{entity } [, \text{entity}] \ldots]
\]

\text{type}\n
Is a data type specifier.

\text{att-ls}\n
Is an optional list of attribute specifiers.

\text{entity}\n
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:

```fortran
REAL, PRIVATE :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the **PUBLIC** and **PRIVATE** statements:

```fortran
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:

```fortran
TYPE, PRIVATE :: DATA
  ...
END TYPE DATA
```

The following example shows a PUBLIC type with PRIVATE components:

```fortran
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

PRIVATE Clause (TU*X only)

Parallel Directive Clause: Declares specified variables to be private to each thread in a team.

Syntax

PRIVATE (list)

list
Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The following occurs when variables are declared in a PRIVATE clause:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the
construct and the corresponding shared variable is undefined on exit from a parallel construct.

- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

For Compaq Fortran parallel directives, LOCAL is an alternative spelling for PRIVATE.

**PROCESSORS_SHAPE**

**Inquiry Intrinsic Function (Specific):** Returns the shape of an implementation-dependent hardware processor array.

This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is provided for compatibility with High Performance Fortran.

**Syntax**

```plaintext
result = PROCESSORS_SHAPE ()
```

**Results:**

The result is a rank-one array of size zero.

**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**

```plaintext
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 a scalar if dim is omitted or array has rank one.

The following rules apply if dim is omitted:

- If PRODUCT(array) is specified, the result is the product of all elements of array. If array has size zero, the result is 1.
- If PRODUCT(array, MASK=mask) is specified, the result is the product of all elements of array corresponding to true elements of mask. If array has size zero, or every element of mask has the value .FALSE., the result is 1.

The following rules apply if dim is specified:

- If array has rank one, the value is the same as PRODUCT(array [,MASK=mask]).
- An array result has a rank that is one less than array, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of array.
- The value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of PRODUCT(array, dim [,mask]) is equal to PRODUCT(array (s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK=mask (s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

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

\[
\begin{bmatrix}
1 & 4 & 7 \\
2 & 3 & 5
\end{bmatrix}
\]
PRODUCT (A, DIM=1) returns the value (2, 12, 35), which is the product of all elements in each column. 2 is the product of 1 * 2 in column 1. 12 is the product of 4 * 3 in column 2, and so forth.

PRODUCT (A, DIM=2) returns the value (28, 30), which is the product of all elements in each row. 28 is the product of 1 * 4 * 7 in row 1. 30 is the product of 2 * 3 * 5 in row 2.

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.

<table>
<thead>
<tr>
<th>Resultant array element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>result(1, 1)</td>
<td>array(1, 1, 1) * array(2, 1, 1)</td>
</tr>
<tr>
<td>result(2, 1)</td>
<td>array(1, 2, 1) * array(2, 2, 1)</td>
</tr>
<tr>
<td>result(1, 2)</td>
<td>array(1, 1, 2) * array(2, 1, 2)</td>
</tr>
<tr>
<td>result(2, 2)</td>
<td>array(1, 2, 2) * array(2, 2, 2)</td>
</tr>
</tbody>
</table>

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.

The **END** statement is the only required part of a program. If a name follows the **END** statement, it must be the same as the name specified in the **PROGRAM** statement.

The program name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A main program must not reference itself (either directly or indirectly).

**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

General Compiler Directive: Modifies several characteristics of a common block.

Syntax

cDEC$ 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 (including Windows 2000) and Windows 9* systems, and 0 through 4 on Tru64 UNIX and Linux 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:
### 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 characteristics of common blocks and how they can be modified by **PSECT**:

<table>
<thead>
<tr>
<th>Default Characteristics</th>
<th>PSECT Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relocatable</td>
<td>None</td>
</tr>
<tr>
<td>Overlaid</td>
<td>None</td>
</tr>
<tr>
<td>Global Scope</td>
<td>Global or local scope</td>
</tr>
<tr>
<td>Not executable</td>
<td>None</td>
</tr>
<tr>
<td>Not multilanguage</td>
<td>Multilanguage or not multilanguage</td>
</tr>
<tr>
<td>Writable</td>
<td>Writable or not writable</td>
</tr>
<tr>
<td>Readable</td>
<td>None</td>
</tr>
<tr>
<td>No protection</td>
<td>None</td>
</tr>
</tbody>
</table>

---

### Default Characteristics of Common Blocks and How They Can Be Modified by PSECT

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Equivalent to val</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>0</td>
</tr>
<tr>
<td>WORD</td>
<td>1</td>
</tr>
<tr>
<td>LONG</td>
<td>2</td>
</tr>
<tr>
<td>QUAD</td>
<td>3</td>
</tr>
<tr>
<td>OCTA</td>
<td>4</td>
</tr>
<tr>
<td>PAGE (VMS only)¹</td>
<td>16</td>
</tr>
</tbody>
</table>

¹ On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, this keyword produces an error.
PSECTIONS (TU*X only)

Compaq Fortran Parallel Compiler Directive: Specifies a block of code to be divided among threads in a team (a worksharing area). This is the same as the OpenMP Fortran API SECTIONS directive with the following exceptions:

- No REDUCTION clause or LASTPRIVATE clause is permitted.
- LOCAL is permitted as an alternative spelling for the PRIVATE clause.

See Also: Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

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:

\[ \text{type, } [\text{att-\textit{ls}},] \ \text{PUBLIC [}, \ \text{att-\textit{ls}}] \ :: \ \text{entity [}, \ \text{entity} \ ... \]

Statement:

\[ \ \text{PUBLIC [}::\] \ \text{entity [}, \ \text{entity} \ ... \]

\textit{type}
Is a data type specifier.

\textit{att-\textit{ls}}
Is an optional list of attribute specifiers.

\textit{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 through 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 VECTRLEN
```

PURE 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
The following variables have restricted use in pure procedures (and any internal procedures):

- Global variables
- Dummy arguments with \texttt{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 \texttt{INTENT} (OUT), \texttt{INTENT} (INOUT), or the \texttt{POINTER} attribute
  - An index variable in a \texttt{DO} or \texttt{FORALL} statement, or an implied-do clause
  - The variable in an \texttt{ASSIGN} statement
  - An input item in a \texttt{READ} statement
  - An internal file unit in a \texttt{WRITE} statement
  - An object in an \texttt{ALLOCATE}, \texttt{DEALLOCATE}, or \texttt{NULLIFY} statement
  - An IOSTAT or SIZE specifier in an I/O statement, or the STAT specifier in a \texttt{ALLOCATE} or \texttt{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 \texttt{READ} or \texttt{WRITE} statement whose I/O unit is an external file unit number or *)
- A \texttt{PAUSE} statement
- A \texttt{STOP} statement

A pure procedure can be used in contexts where other procedures are restricted; for example:

- It can be called directly in a \texttt{FORALL} statement or be used in the mask expression of a \texttt{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:

```fortran
PURE FUNCTION DOUBLE(X)
REAL, INTENT(IN) :: X
DOUBLE = 2 * X
END FUNCTION DOUBLE
```

The following shows another example:

```fortran
PURE INTEGER FUNCTION MANDELBROT(X)
COMPLEX, INTENT(IN) :: X
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:

```fortran
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:

```fortran
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**

**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**

```fortran
! 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
```
**QCMPLX (VMS, U*X)**

**Elemental Intrinsic Function (Generic):** Converts the argument to COMPLEX (16) type. This function must not be passed as an actual argument.

**Syntax**

\[
\text{result} = \text{QCMPLX}(x[, y])
\]

- **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 COMPLEX(16) (or COMPLEX*32).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If \(y\) is not specified and \(x\) is complex, the result value is CMPLX (REAL\((x)\), AIMAG\((x)\)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

\(\text{QCMPLX}(x, y)\) has the complex value whose real part is \(\text{REAL}(x, \text{kind}=16)\) and whose imaginary part is \(\text{REAL}(y, \text{kind}=16)\).

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** CMPLX, DCMPLX, FLOAT, INT, IFIX, REAL, SNGL

**Examples**

QCMPLX (-3) has the value (-3.0Q0, 0.0Q0).

QCMPLX (4.1, 2.3) has the value (4.1Q0, 2.3Q0).
**Elemental Intrinsic Function (Generic):** Converts a number to quad precision (REAL(16)) type.

**Syntax**

\[
\text{result} = \text{QEXT} (a)
\]

\(a\)

(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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(4)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(8)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>QEXT</td>
<td>REAL(4)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>QEXTD</td>
<td>REAL(8)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>COMPLEX(4)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>COMPLEX(8)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td></td>
<td>COMPLEX(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>
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, 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).

**QREAL (VMS, U*X)**

**Elemental Intrinsic Function (Specific):** Converts the real part of a COMPLEX(16) argument to REAL(16) type. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

**Syntax**

```
result = QREAL (a)
```
(Input) Must be of type COMPLEX(16) (or COMPLEX*32).

Results:
The result type is double precision real (REAL(16) or REAL*16).

Compatibility
CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: REAL, DREAL

Examples
QREAL ((2.0q0, 3.0q0)) has the value 2.0q0.

QSORT

Portability Subroutine: Performs a quick sort on an array of rank one.

Module: USE DFPOR

Syntax

CALL QSORT (array, len, isize, compar)

array
(Input) Any type. One-dimensional array to be sorted.
If the data type does not conform to one of the predefined interfaces for
QSORT, you may have to create a new interface (see below).

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 declaration 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 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 or a type that is not in the predefined interfaces, you must include an overload for the generic subroutine QSORT. Examples of how to do this are in the portability module's source file, DFPOR.T.F90, located in your \DF98\INCLUDE subdirectory.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**Example**

```fortran
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

\[
\text{result} = \text{RADIX} (x)
\]

\[x\]

(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, \(\text{RADIX} (X)\) has the value 2.

**RAISEQQ**

Run-Time Function: Sends a signal to the executing program.

Module: USE DFLIB

Syntax

\[
\text{result} = \text{RAISEQQ} (sig)
\]

\[sig\]

(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
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.

This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.

Syntax

$$result = \text{RAN} \ (i)$$

$i$
(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 $0 \leq x < 1$.

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 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, 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.

<table>
<thead>
<tr>
<th>Value of iflag</th>
<th>Selection process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The generator is restarted and the first random value is selected.</td>
</tr>
<tr>
<td>0</td>
<td>The next random number in the sequence is selected.</td>
</tr>
<tr>
<td>Otherwise</td>
<td>The generator is reseeded using iflag, restarted, and the first random value is selected.</td>
</tr>
</tbody>
</table>
When \texttt{RAND} is called without an argument, \texttt{iflag} is assumed to be 0.

There is no difference between \texttt{RAND} and \texttt{RANDOM}. Both functions are included to ensure portability of existing code that references one or both of them. The intrinsic functions \texttt{RANDOM\_NUMBER} and \texttt{RANDOM\_SEED} provide the same functionality.

\textbf{Note:} Because Visual Fortran offers an intrinsic subroutine also called \texttt{RANDOM} in the default library, the only way to access this portability function is with the \texttt{USE DFPORT} statement. Without it, you can only access the default subroutine.

\textbf{Compatibility}

\texttt{CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB}

\textbf{See Also:} \texttt{RANDOM\_NUMBER, RANDOM\_SEED, RANDOM}

\section*{RANDOM}

\textbf{Run-Time Subroutine:} Returns a pseudorandom number greater than or equal to zero and less than one from the uniform distribution.

\textbf{Module:} USE DFLIB

\textbf{Syntax}

\begin{verbatim}
CALL RANDOM (ranval)
\end{verbatim}

\begin{verbatim}
ranval (Output) REAL(4). Pseudorandom number, 0 <= ranval < 1, from the uniform distribution.
\end{verbatim}

A given seed always produces the same sequence of values from \texttt{RANDOM}.

If \texttt{SEED} is not called before the first call to \texttt{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 \texttt{SEED} before the first call to \texttt{RANDOM}.

All the random procedures (\texttt{RANDOM, RAN}, and \texttt{RANDOM\_NUMBER}, and the portability functions \texttt{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:

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:

```fortran
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:

```fortran
REAL x, array1 (5, 5)
CALL RANDOM_SEED()
CALL RANDOM_NUMBER(x)
CALL RANDOM_NUMBER(array1)
```

Consider also the following:

```fortran
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**

```fortran
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:

```fortran
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:

```fortran
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 \( 0.0 \leq y(n + 1) < 1 \).

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**

\[
\text{result} = \text{RANGE}(x)
\]

\(x\) (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 \(\text{INT}(\log_{10}(\text{HUGE}(x)))\). For information on the integer model, see [Model for Integer Data](#).

For a real or complex argument, the result has the value \(\text{INT}(\min(\log_{10}(\text{HUGE}(x)), -\log_{10}(\text{TINY}(x))))\). 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 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 [, istat] [, 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 \textit{nm\_group} is not the second I/O specifier.

\textit{nml}
Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist I/O.

\textit{rec}
Is the cell number of a record to be accessed directly. Optionally prefaced by REC=.

\textit{key} (VMS only)
Is a key specifier (\texttt{KEY}[\textit{con}]=\textit{value}).

\textit{keyid} (VMS only)
Is a key-of-reference specifier (\texttt{KEYID}=\textit{kn}).

\textit{iunit}
Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if \textit{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 \texttt{READ} statement is an expression that calls a function, that function must not execute an I/O statement or the \texttt{EOF} intrinsic function, because the results are unpredictable.

If I/O is to or from a formatted device, \textit{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, \textit{io-list} can contain either derived type components or a derived type variable.

The \texttt{READ} statement can disrupt the results of certain graphics text functions (such as \texttt{SETTEXTWINDOW}) that alter the location of the cursor. You can avoid the problem by getting keyboard input with the \texttt{GETCHARQQ} function and echoing the keystrokes to the screen using \texttt{OUTTEXT}. Alternatively, you can use \texttt{SETTEXTPOSITION} to control cursor location.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{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

Statement: Specifies the REAL data type.

Syntax

```
REAL
REAL([KIND=]n)
REAL*4
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 \texttt{REAL}(8). No kind parameter is permitted for data declared with type \texttt{DOUBLE PRECISION}.

To change the default kind value, use the \texttt{/real_size} compiler option or the \texttt{cDEC$ \texttt{REAL:8}} directive.

\texttt{REAL}(4) and \texttt{REAL*4} (single precision) are the same data type. \texttt{REAL}(8), \texttt{REAL*8}, and \texttt{DOUBLE PRECISION} are the same data type.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \texttt{DOUBLE PRECISION}, \texttt{REAL directive}, \texttt{Real Data Types}, \texttt{General Rules}
for Real Constants, REAL(4) Constants, REAL(8) or DOUBLE PRECISION Constants

Examples

Entity-oriented examples are:

```fortran
MODULE DATDECLARE
    REAL (8), OPTIONAL :: testval=50.d0
    REAL, SAVE :: a(10), b(20,30)
    REAL, PARAMETER :: x = 100.
END MODULE
```

Attribute-oriented examples are:

```fortran
MODULE DATDECLARE
    REAL (8) testval=50.d0
    REAL x, a(10), b(20,30)
    OPTIONAL testval
    SAVE a, b
    PARAMETER (x = 100.)
END MODULE
```

REAL Directive

General Compiler Directive: Specifies the default real kind.

Syntax

```fortran
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
REAL Directive

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.

<table>
<thead>
<tr>
<th>Specific Name 1</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(1)</td>
<td>REAL(4)</td>
<td></td>
</tr>
<tr>
<td>FLOATI</td>
<td>INTEGER(2)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>FLOAT 2, 3</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
</tr>
</tbody>
</table>
REAL Function

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.

RECORD

Statement: Declares a record structure as an entity with a name.

Syntax

```
RECORD /structure-name/record-namelist
[ [, /structure-name/record-namelist] ]
```

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
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):

- $\textit{GFILLINTERIOR}$ - Draws a solid figure using the current color and fill mask.
- $\textit{GBORDER}$ - 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**

```fortran
! 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
```

**REDUCTION (TU*X only)**

**Parallel Directive Clause:** Performs a commutative reduction operation on the specified variables.

**Syntax**

```
REDUCTION ( operator | intrinsic : list )
```

*operator*

Is one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV.

*intrinsic*

Is one of the following: **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.

*list*

Is the name of one or more scalar variables or intrinsic type or common
blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in list is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the operator (see the table below).

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value; the partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct.

However, if the REDUCTION clause is used in a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all the threads complete the REDUCTION clause.

The REDUCTION clause must be used in a region or worksharing construct where the reduction variable is used only in a reduction statement having one of the following forms:

\[
\begin{align*}
x &= x \text{ operator expr} \\
x &= \text{ expr operator x (except for subtraction)} \\
x &= \text{ intrinsic (x, expr)} \\
x &= \text{ intrinsic (expr, x)}
\end{align*}
\]

Some reductions can be expressed in other forms. For instance, a MAX reduction can be expressed as follows:

\[
\text{IF (x .LT. expr) x = expr}
\]

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator you specify in the REDUCTION clause matches the reduction operation.

The following table lists the operators and intrinsics and their initialization values. The actual initialization value will be consistent with the data type of the
reduction variable.

**Initialization Values for REDUCTION Operators and Ininsics (TU*X only)**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initialization Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.FALSE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Initialization Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>Smallest representable number</td>
</tr>
<tr>
<td>MIN</td>
<td>Largest representable number</td>
</tr>
<tr>
<td>IAND</td>
<td>All bits on</td>
</tr>
<tr>
<td>IOR</td>
<td>0</td>
</tr>
<tr>
<td>IEOR</td>
<td>0</td>
</tr>
</tbody>
</table>

If a directive allows reduction clauses, the number you can specify is not limited. However, each variable name can appear in only one of the clauses.

**%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**

\[
\text{result} = \%\text{REF} \ (a)
\]

\textit{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 Compaq Fortran defaults for argument passing, and the allowed uses of `%REF`:

<table>
<thead>
<tr>
<th>Actual Argument Data Type</th>
<th>Default</th>
<th>%REF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressions:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Integer</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>REAL(4)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>REAL(8)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>REAL(16) ¹</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>COMPLEX(4)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>COMPLEX(8)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>COMPLEX(16) ¹</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Character</td>
<td>See table note ²</td>
<td>Yes</td>
</tr>
<tr>
<td>Hollerith</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Aggregate ³</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Derived</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Array Name:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Character</td>
<td>See table note ²</td>
<td>Yes</td>
</tr>
<tr>
<td>Aggregate ³</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>Derived</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Procedure Name:</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ REAL(16) is a 32-bit real number.
² See table note ² for character arguments.
³ Aggregate includes character and numeric arrays.
The %REF, %VAL, and %DESCR functions override related cDEC$ ATTRIBUTE settings.

See Also: CALL, %VAL, %DESCR

Example

```fortran
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.

Note that on Windows NT (including Windows 2000) and Windows 9* systems, the option `/iface` determines how the character argument for variable B is passed.

**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**

```fortran
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 in the Programmer's Guide.

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).

<table>
<thead>
<tr>
<th>Index</th>
<th>Color</th>
<th>Index</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$BLACK</td>
<td>8</td>
<td>$GRAY</td>
</tr>
<tr>
<td>1</td>
<td>$BLUE</td>
<td>9</td>
<td>$LIGHTBLUE</td>
</tr>
<tr>
<td>2</td>
<td>$GREEN</td>
<td>10</td>
<td>$LIGHTGREEN</td>
</tr>
<tr>
<td>3</td>
<td>$CYAN</td>
<td>11</td>
<td>$LIGHTCYAN</td>
</tr>
<tr>
<td>4</td>
<td>$RED</td>
<td>12</td>
<td>$LIGHTRED</td>
</tr>
<tr>
<td>5</td>
<td>$MAGENTA</td>
<td>13</td>
<td>$LIGHTMAGENTA</td>
</tr>
<tr>
<td>6</td>
<td>$BROWN</td>
<td>14</td>
<td>$YELLOW</td>
</tr>
<tr>
<td>7</td>
<td>$WHITE</td>
<td>15</td>
<td>$BRIGHTWHITE</td>
</tr>
</tbody>
</table>

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:

```
  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 G G G G G G R R R R R R R R
```

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 (see Warning below).

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code, such as:

- **EACCESS**: 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.

**Warning:** This routine can cause data to be lost. If the file specified in *to* already exists, **RENAME** deletes the pre-existing file.

It is possible to rename a file to itself without error.

The paths can 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**

```fortran
use dfport
integer(4) istatus
character*12 old_name, new_name
print *, "Enter file to rename: "
read *, old_name
print *, "Enter new name: ", new_name
read *, new_name
ISTATUS = RENAME (old_name, new_name)
```

**RENAMEFILEQQ**

**Run-Time Function:** Renames a file.
RENAMEFILEQQ

Page 35 of 48

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$ACCES - Permission denied. The file's permission setting does not
allow the specified access.
ERR$EXIST - The file already exists.
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, GETLASTERRORQQ
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**

\[
\text{result} = \text{REPEAT} \left(\text{string}, \text{ncopies}\right)
\]

- **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 \(\text{ncopies} \times \text{LEN} (\text{string})\). The kind parameter is the same as string. The value of the result is the concatenation of \(\text{ncopies}\) 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:

```fortran
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**

\[
\text{result} = \text{RESHAPE} \left(\text{source}, \text{shape} [, \text{pad}] [, \text{order}]\right)
\]
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 \textbf{PRODUCT (shape)} if \textit{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 \((1,2,\ldots,n)\), where \(n\) is the size of shape. If order is omitted, it is assumed to be \((1,2,\ldots,n)\).

**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

\[
\begin{bmatrix}
3 & 5 & 7 \\
4 & 6 & 8
\end{bmatrix}.
\]

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value

\[
\begin{bmatrix}
3 & 4 & 5 & 6 \\
7 & 8 & 1 & 1
\end{bmatrix}.
\]

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:

```fortran
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:

```fortran
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
```

**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 obsolete feature in Fortran 95 and Fortran 90.)

**Rules and Behavior**

When a *RETURN* statement is executed in a function subprogram, control is transferred to the referencing statement in the calling program unit.

When a *RETURN* statement is executed in a subroutine subprogram, control is transferred to the first executable statement following the *CALL* statement that invoked the subroutine, or to the alternate return (if one is specified).

**Compatibility**
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'
```
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.

On Tru64 UNIX and Linux systems, a REWIND statement must not be specified for a file that is open for direct access. On Windows NT (including Windows 2000) and Windows 9* systems, it is allowed if 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).

\texttt{err}

Is a branch specifier (ERR=label) if an error condition occurs.

\texttt{io-list}

Is an I/O list.

In the REWRITE statement, data (translated if formatted; untranslated if unformatted) is written to the current (existing) record in one of the following types of external files:

- On Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux systems: In a file with direct access.
- On OpenVMS systems: In all types of files. In sequential files, the current record and new record must be the same length.

The current record is the last record accessed by a preceding, successful sequential, or direct-access \texttt{READ} statement.

Between a \texttt{READ} and \texttt{REWRITE} statement, you should not specify any other I/O statement (except \texttt{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 \texttt{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 \texttt{RECL} in an \texttt{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.

\textbf{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:

\begin{verbatim}
  REWRITE (3, 10, ERR=99) NAME, ,AGE, BIRTH
  10   FORMAT (A16, I2, A8)
\end{verbatim}

\textbf{RGBTOINTEGER}

\textbf{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**

```plaintext
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

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**

```plaintext
! 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 occurrence of substr 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**

```plaintext
result = RRSPACING (x)
```

**x**

(Input) Must be of type real.

**Results:**

The result type is the same as \( x \). The result has the value \( |x \times b^{-e}| \times b^{p} \). 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 \( \times 2^{24} \).

The following shows another example:

```plaintext
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 (Generic):** 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](/).

```plaintext
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:**

\[ type, [att-Is,] \text{SAVE} [, att-Is] :: [object [, object] ...] \]

**Statement:**

\[ \text{SAVE} [object [, object] ...] \]

*type*

Is a data type specifier.

*att-Is*

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 Compaq 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, Compaq 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, MODULE, MODULE procedure, Type Declarations, Compatible attributes.

Examples

The following example shows a type declaration statement specifying the SAVE attribute:

```fortran
SUBROUTINE TEST()
    REAL, SAVE :: X, Y
```

The following is an example of the SAVE statement:

```fortran
SAVE A, /BLOCK_B/, C, /BLOCK_D/, E
```

The following shows another example:

```fortran
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

\[
\text{result} = \text{SAVEIMAGE}\ (\text{filename}, \text{ulxcoord}, \text{ulycoord}, \text{lrxcoord}, \text{lrycoord})
\]
\[
\text{result} = \text{SAVEIMAGE\_W}\ (\text{filename}, \text{ulwxcoord}, \text{ulwycoord}, \text{lrwxcoord}, \text{lrwycoord})
\]

\text{filename}
(Input) Character*. Path of the bitmap file.

\text{ulxcoord, ulycoord}
(Input) INTEGER(4). Viewport coordinates for upper-left corner of the screen image to be captured.

\text{lrxcoord, lrycoord}
(Input) INTEGER(4). Viewport coordinates for lower-right corner of the screen image to be captured.

\text{ulwxcoord, ulwycoord}
(Input) REAL(8). Window coordinates for upper-left corner of the screen image to be captured.

\text{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 \text{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 \text{filename}. The image is stored with a palette containing the colors displayed on the screen.

\text{SAVEIMAGE} defines the bounding rectangle in viewport coordinates. \text{SAVEIMAGE\_W} defines the bounding rectangle in window coordinates.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: \text{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 \(x \times b^i\). 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:

```fortran
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
```
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(*,'(I2)') row
CALL OUTTEXT( string )
END

**SCWRQQ (x86 only)**

**Run-Time Subroutine:** Returns the floating-point processor control word. This routine is only available on x86 processors.

**Module:** USE DFLIB

**Syntax**

```fortran
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.

This is a specific function that has no generic function associated with it. It must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.

Syntax

\[
\text{result} = \text{SECNDS}(x)
\]

\(x\)

(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 \text{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 \text{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

\textbf{SECNDS Portability Routine}

\textbf{Portability Function:} Returns the number of seconds that have elapsed since
midnight, less the value of its argument.

**Module: USE DFPORT**

**Syntax**

\[
\text{result} = \text{SECNDS}(r)
\]

\(r\)

(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**

```fortran
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.'
```

**SECTIONS (TU*X only)**

**OpenMP Parallel Compiler Directive:** Specifies one or more blocks of code that must be divided among threads in the team. Each section is executed once by a thread in the team.

**Syntax**

\[
\text{c$OMP SECTIONS} \ [\text{clause}[,\text{clause}]] \ ... \]
\[
\text{[c$OMP SECTION]} \]
\[
\text{block}
\]
\[
\text{[c$OMP SECTION]}
\]
\[
\text{block} \]
\[
... \]

```fortran
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.'
```
c$OMP END SECTIONS [NOWAIT]

c
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

clause
Is one of the following:

- FIRSTPRIVATE (list)
- LASTPRIVATE (list)
- PRIVATE (list)
- REDUCTION( operator | intrinsic : list)

block
Is a structured block (section) of statements or constructs. Any constituent section must also be a structured block.

You cannot branch into or out of the block.

Rules and Behavior

Each section of code is preceded by a SECTION directive, although the directive is optional for the first section. The SECTION directives must appear within the lexical extent of the SECTIONS and END SECTIONS directive pair.

The last section ends at the END SECTIONS directive. Threads that complete execution of their SECTIONs encounter an implied barrier at the END SECTIONS directive unless NOWAIT is specified.

SECTIONS directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

c$OMP PARALLEL
c$OMP SECTIONS
c$OMP SECTION
SECTIONS (TU*X only)

CALL XAXIS
c$OMP SECTION
CALL YAXIS
c$OMP SECTION
CALL ZAXIS
c$OMP END SECTIONS
c$OMP END PARALLEL

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

```fortran
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**

```fortran
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](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**
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

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

\[
\text{result} = \text{SETBKCOLOR} \left( \text{color} \right)
\]

\[
\text{color}
\]

(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

\[
\text{result} = \text{SETBKCOLORRGB}(\text{color})
\]

\[
\text{color} \quad \text{(Input) INTEGER(4). RGB color value to set the background color to. Range and result depend on the system's display adapter.}
\]

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>E B B E B B B B</td>
<td>G G G G G G G G</td>
<td>R R R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

![Diagram](image)

**SETCOLOR**

**Graphics Function:** Sets the current graphics color index.

**Module:** USE DFLIB

**Syntax**

```fortran
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**, **SETPIEXLS**
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

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.
**SETCOLORRGB** 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, REMAPPACTERRGB, SETPIXELRGB, SETPIXELSRGB

**Example**

```plaintext
! 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 x86 processors.

**Module:** USE DFLIB

**Syntax**

```plaintext
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:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Hex value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCW$MCW_IC</td>
<td>#1000</td>
<td><strong>Infinity control mask</strong></td>
</tr>
<tr>
<td>FPCW$AFFINE</td>
<td>#1000</td>
<td>Affine infinity</td>
</tr>
<tr>
<td>FPCW$PROJECTIVE</td>
<td>#0000</td>
<td>Projective infinity</td>
</tr>
<tr>
<td>FPCW$MCW_PC</td>
<td>#0300</td>
<td><strong>Precision control mask</strong></td>
</tr>
<tr>
<td>FPCW$64</td>
<td>#0300</td>
<td>64-bit precision</td>
</tr>
<tr>
<td>FPCW$53</td>
<td>#0200</td>
<td>53-bit precision</td>
</tr>
<tr>
<td>FPCW$24</td>
<td>#0000</td>
<td>24-bit precision</td>
</tr>
<tr>
<td>FPCW$MCW_RC</td>
<td>#0C00</td>
<td><strong>Rounding control mask</strong></td>
</tr>
<tr>
<td>FPCW$CHOP</td>
<td>#0C00</td>
<td>Truncate</td>
</tr>
<tr>
<td>FPCW$UP</td>
<td>#0800</td>
<td>Round up</td>
</tr>
<tr>
<td>FPCW$DOWN</td>
<td>#0400</td>
<td>Round down</td>
</tr>
<tr>
<td>FPCW$NEAR</td>
<td>#0000</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>FPCW$MCW_EM</td>
<td>#003F</td>
<td><strong>Exception mask</strong></td>
</tr>
<tr>
<td>FPCW$INVALID</td>
<td>#0001</td>
<td>Allow invalid numbers</td>
</tr>
<tr>
<td>FPCW$DENORMAL</td>
<td>#0002</td>
<td>Allow denormals (very small numbers)</td>
</tr>
<tr>
<td>FPCW$ZERODIVIDE</td>
<td>#0004</td>
<td>Allow divide by zero</td>
</tr>
<tr>
<td>FPCW$OVERFLOW</td>
<td>#0008</td>
<td>Allow overflow</td>
</tr>
<tr>
<td>FPCW$UNDERFLOW</td>
<td>#0010</td>
<td>Allow underflow</td>
</tr>
<tr>
<td>FPCW$INEXACT</td>
<td>#0020</td>
<td>Allow inexact precision</td>
</tr>
</tbody>
</table>
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`, `SCWROQ`, `CLEARSTATUSFPQQ`

**Example**

```fortran
USE DFLIB
INTEGER(2) status, control, controlo

CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!
controlo = control
!
control = control .AND. #0000
!
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

\[
\text{result} = \text{SETENVQQ} (\text{varname}=\text{value})
\]

\[
\text{varname} = \text{value}
\]

(Input) Character*(*) String containing both the name and the value of the variable to be added or modified. Must be in the form: \(\text{varname} = \text{value}\), where \(\text{varname}\) is the name of an environment variable and \(\text{value}\) 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.

\text{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 \text{value} so that trailing blanks are preserved. For example, the string \text{PATH= } = \text{sets value to ''}.

You can use \text{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.

\text{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 \text{SETENVQQ} to any child process created by \text{RUNQQ}. These child processes get new variables and/or values added by \text{SETENVQQ}.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \text{GETENVQQ}

Example

\begin{verbatim}
USE DFLIB
LOGICAL(4) success
\end{verbatim}
success = SETENVQQ("PATH=c:\mydir\tmp")
success = &
SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")
END

SETENVQQ

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

\[ \text{result} = \text{SET\_EXponent} \ (x, i) \]

\(x\)
(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 \(x \times b^{i \times e}\). 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**

\[
\text{result} = \text{SETFILEACCESSQQ} (\text{filename}, \text{access})
\]

**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**

```fortran
USE DFLIB
INTEGER(4) permit
LOGICAL(4) result

permit = 0    ! clear permit
permit = IOR(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**

```fortran
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,
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. It can be one of the following:

- **ERR$ACCES** - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- **ERR$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.
- **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.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** `PACKTIMEQQ, UNPACKTIMEQQ, GETLASTERRORQQ`

**Example**

```fortran
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.

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Value in mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>● ● ● ● ● ● ● ●</td>
<td><code>mask(1) = #93</code></td>
</tr>
<tr>
<td>● ● ● ● ● ● ● ●</td>
<td><code>mask(2) = #C9</code></td>
</tr>
<tr>
<td>○ ● ● ● ● ● ● ●</td>
<td><code>mask(3) = #64</code></td>
</tr>
<tr>
<td>● ● ● ● ● ● ● ●</td>
<td><code>mask(4) = #B2</code></td>
</tr>
<tr>
<td>● ● ● ● ● ● ● ●</td>
<td><code>mask(5) = #59</code></td>
</tr>
<tr>
<td>○ ● ● ● ● ● ● ●</td>
<td><code>mask(6) = #2C</code></td>
</tr>
<tr>
<td>● ● ● ● ● ● ● ●</td>
<td><code>mask(7) = #96</code></td>
</tr>
<tr>
<td>○ ● ● ● ● ● ● ●</td>
<td><code>mask(8) = #4B</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bit</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>

**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.

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**

\[
\text{result} = \text{SETFONT}(\text{options})
\]

\[
\text{options}
\]

(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 \(nx\) 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 \(\text{options}\). 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 \(\text{options}\) argument consists of letter codes, as follows, that describe the desired font. The \(\text{options}\) parameter is not case sensitive or position sensitive.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>fontname</td>
</tr>
<tr>
<td>h</td>
<td>Character height, where (y) is the number of pixels.</td>
</tr>
<tr>
<td>w</td>
<td>Select character width, where (x) is the number of pixels.</td>
</tr>
<tr>
<td>f</td>
<td>Select only a fixed-space font (do not use with the (p) characteristic).</td>
</tr>
<tr>
<td>p</td>
<td>Select only a proportional-space font (do not use with the (f) characteristic).</td>
</tr>
<tr>
<td>v</td>
<td>Select only a vector-mapped font (do not use with the (r) characteristic). In Windows NT (including Windows 2000), 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.</td>
</tr>
<tr>
<td>r</td>
<td>Select only a raster-mapped (bitmapped) font (do not use with the (v) characteristic). In Windows NT (including Windows 2000), Courier, Helvetica, and Palatino are examples of raster-mapped fonts, also called screen fonts. True Type fonts are not raster-mapped.</td>
</tr>
</tbody>
</table>
You can specify as many options as you want, except with `nx`, 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 `nx`.

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 `nx` parameter, `SETFONT` ignores any other specified options and supplies only the font number corresponding to `x`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>e</code></td>
<td>Select the bold text format. This parameter is ignored if the font does not allow the bold format.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Select the underline text format. This parameter is ignored if the font does not allow underlining.</td>
</tr>
<tr>
<td><code>i</code></td>
<td>Select the italic text format. This parameter is ignored if the font does not allow italics.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>Select the font that best fits the other parameters specified.</td>
</tr>
<tr>
<td><code>nx</code></td>
<td>Select font number <code>x</code>, where <code>x</code> is less than or equal to the value returned by the <code>INITIALIZEFONTS</code> function.</td>
</tr>
</tbody>
</table>
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**

```fortran
! 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**

```fortran
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**

```plaintext
! 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 closely 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:

<table>
<thead>
<tr>
<th>QuickWin Mask</th>
<th>Internal Windows Style</th>
<th>Selection Criteria</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>PS_SOLID</td>
<td>16 bits on</td>
<td>____________</td>
</tr>
<tr>
<td>0xEEEE</td>
<td>PS_DASH</td>
<td>11 to 15 bits on</td>
<td>--------------</td>
</tr>
<tr>
<td>0xECEC</td>
<td>PS_DASHDOT</td>
<td>10 bits on</td>
<td>-. -. -. -. -. .</td>
</tr>
<tr>
<td>0xECCC</td>
<td>PS_DASHDOTDOT</td>
<td>9 bits on</td>
<td>-. -. -. -. -. -</td>
</tr>
<tr>
<td>0xAAAA</td>
<td>PS_DOT</td>
<td>1 to 8 bits on</td>
<td>................</td>
</tr>
<tr>
<td>0x0000</td>
<td>PS_NULL</td>
<td>0 bits on</td>
<td></td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Id</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>QWIN$MSG_TERM</td>
<td>&quot;Program terminated with exit code&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_EXITQ</td>
<td>&quot;\nExit Window?&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_FINISHED</td>
<td>&quot;Finished&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_PAUSED</td>
<td>&quot;Paused&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_RUNNING</td>
<td>&quot;Running&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_FILEOPENDLG</td>
<td>&quot;Text Files(<em>.txt), <em>.txt; Data Files (</em>.dat), <em>.dat; All Files(</em>.</em>), <em>.</em>;&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_BMPSAVEDLG</td>
<td>&quot;Bitmap Files(<em>.bmp), <em>.bmp; All Files (</em>.</em>)<em>, <em>.</em>,</em>;&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_INPUTPEND</td>
<td>&quot;Input pending in&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_PASTEINPUTPEND</td>
<td>&quot;Paste input pending&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_MOUSEINPUTPEND</td>
<td>&quot;Mouse input pending in&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_SELECTTEXT</td>
<td>&quot;Select Text in&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_SELECTGRAPHICS</td>
<td>&quot;Select Graphics in&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_PRINTABORT</td>
<td>&quot;Error! Printing Aborted.&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_PRINTLOAD</td>
<td>&quot;Error loading printer driver&quot;</td>
</tr>
<tr>
<td>QWIN$MSG_PRINTNODEFAULT</td>
<td>&quot;No Default Printer.&quot;</td>
</tr>
</tbody>
</table>
Note that QWIN$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 \texttt{SETMESSAGEQQ} with the appropriate \textit{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 \textit{Programmer's Guide}.) To change menu messages, use \texttt{MODIFYMENUSTRINGQQ}.

\textbf{Compatibility}

QUICKWIN GRAPHICS LIB

\textbf{See Also:} \texttt{MODIFYMENUSTRINGQQ}

\textbf{Example}

\begin{verbatim}
USE DFLIB
print*, "Hello"
CALL SETMESSAGEQQ('Changed exit text', QWIN$MSG_EXIT)
\end{verbatim}

\textbf{SETMOUSECURSOR}

\textbf{Graphics Function:} Sets the shape of the mouse cursor for the window in focus.

\textbf{Module:} \texttt{USE DFLIB USE DFWIN}
Syntax

oldcursor = SETMOUSECURSOR (newcursor)

newcursor
(Input) INTEGER(4). A Windows HCURSOR value. For many predefined shapes, LoadCursor(0, shape) is a convenient way to get a legitimate value. See the list of predefined shapes below.

A value of zero causes the cursor not to be displayed.

Results:

The result type is INTEGER(4). This is also an HCURSOR Value. The result is the previous cursor value.

The window in focus at the time SETMOUSECURSOR is called has its cursor changed to the specified value. Once changed, the cursor retains its shape until another call to SETMOUSECURSOR.

In Standard Graphics applications, units 5 and 6 (the default screen input and output units) are always considered to be in focus.

The following predefined values for cursor shapes are available:

<table>
<thead>
<tr>
<th>Predefined Value</th>
<th>Cursor Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDC_APPSTARTING</td>
<td>Standard arrow and small hourglass</td>
</tr>
<tr>
<td>IDC_ARROW</td>
<td>Standard arrow</td>
</tr>
<tr>
<td>IDC_CROSS</td>
<td>Crosshair</td>
</tr>
<tr>
<td>IDC_BEAM</td>
<td>Text I-beam</td>
</tr>
<tr>
<td>IDC_ICON</td>
<td>Obsolete value</td>
</tr>
<tr>
<td>IDC_NO</td>
<td>Slashed circle</td>
</tr>
<tr>
<td>IDC_SIZE</td>
<td>Obsolete value; use IDC_SIZEALL</td>
</tr>
<tr>
<td>IDC_SIZEALL</td>
<td>Four-pointed arrow</td>
</tr>
<tr>
<td>IDC_SIZENESW</td>
<td>Double-pointed arrow pointing northeast and southwest</td>
</tr>
<tr>
<td>IDC_SIZENS</td>
<td>Double-pointed arrow pointing north and south</td>
</tr>
</tbody>
</table>
A LoadCursor must be done on these values before they can be used by 
**SETMOUSECURSOR**.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

**Examples**

! Build as QuickWin

```fortran
    use dflib
    use dfwin
    integer*4  cursor, oldcursor
    open(8, file='user')
    write(8,*) 'The cursor will now be changed to an hour glass shape'
    write(8,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_WAIT)
    oldcursor = SetMouseCursor(cursor)
    read(8,*)
    write(8,*) 'The cursor will now be changed to a cross-hair shape'
    write(8,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_CROSS)
    oldcursor = SetMouseCursor(cursor)
    read(8,*)
    write(8,*) 'The cursor will now be turned off'
    write(8,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(0)
    read(8,*)
    write(8,*) 'The cursor will now be turned on'
    write(8,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(oldcursor)
    read(8,*)
    stop
    end
```

! Build as Standard Graphics or QuickWin

```fortran
    use dflib
    use dfwin
    integer*4  cursor, oldcursor
    write(6,*) 'The cursor will now be changed to an hour glass shape'
    write(6,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_WAIT)
    oldcursor = SetMouseCursor(cursor)
    read(5,*)
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDC_SIZENWSE</td>
<td>Double-pointed arrow pointing northwest and southeast</td>
</tr>
<tr>
<td>IDC_SIZEWE</td>
<td>Double-pointed arrow pointing west and east</td>
</tr>
<tr>
<td>IDC_UPARROW</td>
<td>Vertical arrow</td>
</tr>
<tr>
<td>IDC_WAIT</td>
<td>Hour glass</td>
</tr>
</tbody>
</table>
write(6,*) 'The cursor will now be changed to a cross-hair shape'
cursor = LoadCursor(0, IDC_CROSS)
oldcursor = SetMouseCursor(cursor)
read(5,*)

write(6,*) 'The cursor will now be turned off'
write(6,*) 'Hit <return> to see the next change'
oldcursor = SetMouseCursor(0)
read(5,*)

write(6,*) 'The cursor will now be turned on'
write(6,*) 'Hit <return> to see the next change'
oldcursor = SetMouseCursor(oldcursor)
read(5,*)

stop
derm

**SETPIXEL, SETPIXEL_W**

**Graphics Function:** Sets a pixel at a specified location to the current graphics color index.

**Module:** USE DFLIB

**Syntax**

\[
\text{result} = \text{SETPIXEL} (x, y) \\
\text{result} = \text{SETPIXEL_W} (wx, wy)
\]

\(x, y\)

(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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0</td>
<td>E B B E B B B G G G G G G R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

```plaintext
! 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
DO
```
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, SETPIXELRGB 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0 B B B B B B G G G G G G G G R R R R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
SETTEXTCOLOR

Graphics Function: Sets the current text color index.

Module: USE DFLIB

Syntax

\[
\text{result} = \text{SETTEXTCOLOR}(\text{index})
\]

\text{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.

\text{SETTEXTCOLOR} sets the current text color index. The default value is 15, which is associated with white unless the user remaps the palette. \text{GETTEXTCOLOR} returns the text color index set by \text{SETTEXTCOLOR}. \text{SETTEXTCOLOR} affects text output with \text{OUTTEXT}, \text{WRITE}, and \text{PRINT}.

The background color index is set with \text{SETBKCOLOR} and returned with \text{GETBKCOLOR}. The color index of graphics over the background color is set with \text{SETCOLOR} and returned with \text{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 \text{SETTEXTCOLORRGB}, \text{SETBKCOLORRGB}, and \text{SETCOLORRGB}.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: \text{GETTEXTCOLOR}, \text{REMAPPALETTERGB}, \text{SETCOLOR}, \text{SETTEXTCOLORRGB}

Example

! Build as a Graphics ap.
USE DFLIB
INTEGER(2) oldtc
oldtc = SETTEXTCOLOR(INT2(2)) ! green
WRITE(*,*) "hello, world"
**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 **SETTEXTCOLORRGB**, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 (MSB)</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>0 0 0 0 0 0 0 0</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

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**

```fortran
! 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
```

**SETTEXTCURSOR**

**Graphics Function:** Sets the height and width of the text cursor (the caret) for the window in focus.

**Module:** USE DFLIB

**Syntax**

```fortran
result = SETTEXTCURSOR (newcursor)
```

`newcursor` (Input) INTEGER(2). The leftmost 8 bits specify the width of the cursor, and the rightmost 8 bits specify the height of the cursor. These dimensions can range from 1 to 8, and represent a fraction of the current character cell size. For example:

- #0808 - Specifies the full character cell; this is the default size.
#0108 - Specifies 1/8th of the character cell width, and 8/8th (or all) of the character cell height.

If either of these dimensions is outside the range 1 to 8, it is forced to 8.

**Results:**

The result type is INTEGER(2); it is the previous text cursor value in the same format as `newcursor`.

---

**Note:** After calling `SETTEXTCURSOR`, you must call `DISPLAYCURSOR` ($GCURSORON) to actually see the cursor.

**Compatibility**

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

**See Also:** `DISPLAYCURSOR`

**Example**

```fortran
use dflib
integer(2) oldcur
integer(2) istat
type(rccoord) rc
open(10,file='user')
istat  = displaycursor($GCURSORON)
write(10,*) 'Text cursor is now character cell size, the default.'
read(10,*)
write(10,*) 'Setting text cursor to wide and low.'
oldcur = settextcursor(#0801)
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to high and narrow.'
oldcur = settextcursor(#0108)
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to a dot.'
oldcur = settextcursor(#0101)
istat  = displaycursor($GCURSORON)
read(10,*)
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**

```fortran
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**

\[
\text{result} = \text{SETTIM} \left( ihr, imin, isec, i100th \right)
\]

\[ ihr \]
(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:
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**

```fortran
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**

```fortran
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 finvert 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 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 system 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 $wx_1$ equals $wx_2$ or $wy_1$ equals $wy_2$, **SETWINDOW** fails.

### Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB


### Example

```fortran
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.
END TYPE
```

INTEGER(2) numcolors    ! Number of color indexes.
INTEGER(4) fontsize     ! Size of default font. Set
! to QWIN$EXTENDFONT when
! specifying extended attributes,
! in which case extendfontsize
! sets the font size.
CHARACTER(80) title     ! The window title, a C string.
! The next three parameters provide extended font
! attributes.
CHARACTER(32) extendfontname ! The name of the desired font.
INTEGER(4) extendfontsize  ! Takes the same values as
! fontsize, when fontsize is
! set to QWIN$EXTENDFONT.
INTEGER(4) extendfontattributes ! Font attributes
! such as bold and italic.
END TYPE windowconfig

Results:

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The following values can be used with SETWINDOWCONFIG extended fonts:

<table>
<thead>
<tr>
<th>Style:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>QWIN$EXTENDFONT_NORMAL</td>
<td>Gives no underline, no italic, and a</td>
</tr>
<tr>
<td></td>
<td>font weight of 400 out of 1000.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_UNDERLINE</td>
<td>Gives underlined characters.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_BOLD</td>
<td>Gives a font weight of 700 out of</td>
</tr>
<tr>
<td></td>
<td>1000.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_ITALIC</td>
<td>Gives italic characters.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pitch:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>QWIN$EXTENDFONT_FIXED_PITCH</td>
<td>QuickWin default. Equal character widths.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_VARIABLE_PITCH</td>
<td>Variable character widths.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Font Families:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>QWIN$EXTENDFONT_FF_ROMAN</td>
<td>Variable stroke width, serifed. Times</td>
</tr>
<tr>
<td></td>
<td>Roman, Century Schoolbook, etc.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_FF_SWISS</td>
<td>Variable stroke width, sans-serifed.</td>
</tr>
<tr>
<td></td>
<td>Helvetica, Swiss, etc.</td>
</tr>
</tbody>
</table>
Using QWIN$EXTENDFONT_OEM_CHARSET with the font name 'MS LineDraw'C will get the old DOS-style character set with symbols that can be used to draw lines and boxes. The pitch and font family items can be specified to help guide the font matching algorithms used by CreateFontIndirect, the WIN32 API used by SETWINDOWCONFIG.

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

<table>
<thead>
<tr>
<th>Font Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QWIN$EXTENDFONT_FF_MODERN</td>
<td>QuickWin default. Constant stroke width, serifed or sans-serifed. Pica, Elite, Courier, etc.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_FF_SCRIPT</td>
<td>Cursive, etc.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_FF_DECORATIVE</td>
<td>Old English, etc.</td>
</tr>
<tr>
<td>Character Sets:</td>
<td></td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_ANSI_CHARSET</td>
<td>QuickWin default.</td>
</tr>
<tr>
<td>QWIN$EXTENDFONT_OEM_CHARSET</td>
<td>Use this to get Microsoft LineDraw.</td>
</tr>
</tbody>
</table>
it.

Under Standard Graphics, the application attempts to start 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%numx pixels  = -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 = SETWINDOWMENUUQQ(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_MERGEPPENNOT**
- **$GR2_MERGEPPEN** (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:

```plaintext
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).
When called from \texttt{INITIALSETT\textsc{\texttt{INGS}}}, \texttt{SETWSIZEQQ} behaves slightly differently than when called from a user routine after initialization. See below under \textit{Results}.

\textit{winfo}  
\texttt{(Input)} Derived type \texttt{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 \texttt{qwinfo} is defined in \texttt{DFLIB.F90} as follows:

\begin{verbatim}
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
\end{verbatim}

This function's behavior depends on the value of \texttt{QWINFO\%TYPE}, which can be any of the following:

- \texttt{QWIN$MIN} - Minimizes the window.
- \texttt{QWIN$MAX} - Maximizes the window.
- \texttt{QWIN$RESTORE} - Restores the minimized window to its previous size.
- \texttt{QWIN$SET} - Sets the window's position and size according to the other values in \texttt{qwinfo}.

\textbf{Results:}

The result type is \texttt{INTEGER(4)}. The result is zero if successful; otherwise, nonzero (unless called from \texttt{INITIALSETT\textsc{\texttt{INGS}}}). If called from \texttt{INITIALSETT\textsc{\texttt{INGS}}}, the following occurs:

- \texttt{SETWSIZEQQ} always returns -1.
- Only \texttt{QWIN$SET} will work.

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 \textit{including} any borders, menus, and status bar at the bottom.

\textbf{Compatibility}

\texttt{QUICKWIN GRAPHICS LIB}
See Also: Using QuickWin, GETWSIZEQQ, INITIALSETTINGS

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 [, kind])

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.

kind
    (Optional; input) Must be a scalar integer initialization expression.

Results:

The result is a rank-one integer array whose size is equal to the rank of source. If kind is present, the kind parameter of the result is that specified by kind; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is the shape of source.

The setting of compiler options that specify integer size can affect the result of this function.
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
```

**SHARED (TU*X only)**

Parallel Directive Clause: Specifies variables that will be shared by all the threads in a team.

Syntax

```
SHARED (list)
```

**list**

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be
separated by a comma, and a named common block must appear between slashes (/ /).

All threads within a team access the same storage area for SHARED data.

**SHORT**

**Portability Function:** Converts an INTEGER(4) value into an equivalent INTEGER(2) type.

**Module:** USE DFPORT

**Syntax**

\[
\text{result} = \text{SHORT}(\text{int4})
\]

\[
\text{int4} \\
\text{(Input) INTEGER(4). Value to be converted.}
\]

**Results:**

The result type is INTEGER(2). The result is equal to the lower 16 bits of \( \text{int4} \). If the \( \text{int4} \) 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
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 processor can distinguish between positive and negative real zero and the `/assume:minus0` compiler option is specified, the following occurs:

- If b is positive real zero, the value of the result is |a|.
- If b is negative real zero, the value of the result is -|a|.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IISIGN</td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>ISIGN ¹</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>KISIGN ²</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>SIGN</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>DSIGN</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>QSIGN ³</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

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:

\[ \begin{align*}
    c &= \text{SIGN} (5.2, -3.1) \quad \text{! returns } -5.2 \\
    c &= \text{SIGN} (-5.2, -3.1) \quad \text{! returns } -5.2 \\
    c &= \text{SIGN} (-5.2, 3.1) \quad \text{! returns } 5.2
\end{align*} \]

SIN

Elemental Intrinsic Function (Generic): Produces a sine (with the result in radians).

Syntax

\[
    \text{result} = \text{SIN} \ (x)
\]

\( x \)

(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 \( x \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DSIN</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QSIN 1</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>CSIN 2</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
</tr>
<tr>
<td>CDSIN 3</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
</tr>
<tr>
<td>CQSIN 1</td>
<td>COMPLEX(16)</td>
<td>COMPLEX(16)</td>
</tr>
</tbody>
</table>
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**

\[ \text{result} = \text{SIND} (x) \]

\(x\) (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 \(x\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIND</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DSIND</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QSIND (^1)</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

\(^1\) 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**

\[ \text{result} = \text{SINH} \ (x) \]

\(x\) (Input) Must be of type real.

**Results:**

The result type is the same as \(x\).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINH</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DSINH</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QSINH</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

\(^1\) 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**

\[ \text{result} = \text{SIGNAL} \ (\text{signum, proc, flag}) \]

\(\text{signum}\)

(Input) INTEGER(4). Number of the signal to change. The numbers and symbolic names are listed in a table below.

\(\text{proc}\)
(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` and `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:

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Floating-point error</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Kill process</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>CTRL+C signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Illegal storage access</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Termination request</td>
</tr>
</tbody>
</table>

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

```fortran
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`, `GETEXCEPTIONPTRSSQ`

**Example**

```plaintext
!  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

SINGLE (TU*X only)

OpenMP Parallel Compiler Directive: Specifies that a block of code is to be executed by only one thread in the team.

Syntax

\[
\texttt{c$OMP SINGLE [\textit{clause}][, \textit{clause}] ... ]} \\
\texttt{block} \\
\texttt{c$OMP END SINGLE [NOWAIT]} \\
\]

\texttt{c}
Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

\texttt{clause}
Is one of the following:

- \texttt{FIRSTPRIVATE (list)}
- \texttt{PRIVATE (list)}

\texttt{block}
Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.
Rules and Behavior

Threads in the team that are not executing this directive wait at the **END SINGLE** directive unless NOWAIT is specified.

**SINGLE** directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

See Also: [Parallel Directives for Tru64 UNIX Systems](#), [OpenMP Fortran API Compiler Directives](#) (TU*X only), [Compaq Fortran Parallel Compiler Directives](#) (TU*X only)

Examples

In the following example, the first thread that encounters the **SINGLE** directive executes subroutines **OUTPUT** and **INPUT**:

```
c$OMP PARALLEL DEFAULT(SHARED)
    CALL WORK(X)
c$OMP BARRIER
    CALL OUTPUT(X)
c$OMP SINGLE
    CALL INPUT(Y)
c$OMP END SINGLE
    CALL WORK(Y)
c$OMP END PARALLEL
```

You should not make assumptions as to which thread executes the **SINGLE** section. All other threads skip the **SINGLE** section and stop at the barrier at the **END SINGLE** construct. If other threads can proceed without waiting for the thread executing the **SINGLE** section, you can specify NOWAIT in the **END SINGLE** directive.

**SINGLE PROCESS** (TU*X only)

**Compaq Fortran Parallel Compiler Directive:** Specifies a block of code to be executed by only one thread in a team. This directive is the same as the OpenMP Fortran API **SINGLE** directive except that LOCAL is permitted as an alternative spelling for the PRIVATE clause.

See Also: [Parallel Directives for Tru64 UNIX Systems](#), [Compaq Fortran Parallel Compiler Directives](#) (TU*X only), [OpenMP Fortran API Compiler Directives](#) (TU*X only)

**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

\[
\text{result} = \text{SIZE}(\text{array} [, \ dim] [, \ \text{kind}])
\]

\textit{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 \textit{dim} is present with a value less than the rank of \textit{array}.

\textit{dim}

(Optional; input) Must be a scalar integer with a value in the range 1 to \(n\), where \(n\) is the rank of \textit{array}.

\textit{kind}

(Optional; input) Must be a scalar integer initialization expression.

Results:

The result is a scalar of type integer. If \textit{kind} is present, the kind parameter of the result is that specified by \textit{kind}; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If \textit{dim} is present, the result is the extent of dimension \textit{dim} in \textit{array}; otherwise, the result is the total number of elements in \textit{array}.

The setting of compiler options that specify integer size can affect the result of this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \textbf{SHAPE}

Examples

If B is declared as \(B(2:4, -3:1)\), then \text{SIZE}(B, \text{DIM}=2)\) has the value 5 and \text{SIZE}(B)\) has the value 15.

The following shows another example:

\begin{verbatim}
REAL(8) array (3:10, -1:3)
INTEGER i
i = SIZE(array, DIM = 2) ! returns 5
i = SIZE(array)            ! returns 40
\end{verbatim}
SIZEOF

Inquiry Intrinsic Function (Specific): Returns the number of bytes of storage used by the argument. This is a specific function that has no generic function associated with it. It must not be passed as an actual argument.

Syntax

\[
\text{result} = \text{SIZEOF} \ (x)
\]

\( x \)
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 x86 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

\[
\text{CALL SLEEP} \ (\text{time})
\]

\( time \)
(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
delay  = 4000
duration = 1000
CALL SLEEPQQ(delay)
CALL BEEPQQ(freq, duration)
END

SNGL

See REAL.function.

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type of array</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRT$INTEGER1</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>SRT$INTEGER2</td>
<td>INTEGER(2) or equivalent</td>
</tr>
<tr>
<td>SRT$INTEGER4</td>
<td>INTEGER(4) or equivalent</td>
</tr>
<tr>
<td>SRT$REAL4</td>
<td>REAL(4) or equivalent</td>
</tr>
<tr>
<td>SRT$REAL8</td>
<td>REAL(8) or equivalent</td>
</tr>
</tbody>
</table>

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:

```fortran
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

```fortran
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
```

SPORT_CANCEL_IO

Run-Time Function: Cancels any I/O in progress to the specified port.

Module: USE DFLIB

Syntax

```
irest = SPORT_CANCEL_IO (port)
```

port  
(Input) Integer. The port number.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.
**SPORT_CANCEL_IO**

**Note:** This call also kills the thread that keeps an outstanding read operation to the serial port. This call **must** be done before any of the port characteristics are modified.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

USE DFLIB
INTEGER(4) iresult
iresult = SPORT_CANCEL_IO( 2 )
END

**SPORT_CONNECT**

**Run-Time Function:** Establishes the connection to a serial port and defines certain usage parameters.

**Module:** USE DFLIB

**Syntax**

iresult = **SPORT_CONNECT** (port, [options])

*port*  
(Input) Integer. The port number of connection. The routine will open COMn, where n is the port number specified.

*options*  
(Optional; input) Integer. Defines the connection options. These options define how the *nnn_LINE* routines will work and also effect the data that is passed to the user. If more than one option is specified, the operator .OR. should be used between each option. Options are as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_TOSS_CR</td>
<td>Removes carriage return (CR) characters on input.</td>
</tr>
<tr>
<td>DL_TOSS_LF</td>
<td>Removes linefeed (LF) characters on input.</td>
</tr>
<tr>
<td>DL_OUT_CR</td>
<td>Causes <strong>SPORT_WRITE_LINE</strong> to add a CR to each record written.</td>
</tr>
<tr>
<td>DL_OUT_LF</td>
<td>Causes <code>SPORT_WRITE_LINE</code> to add a LF to each record written.</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>DL_TERM_CR</td>
<td>Causes <code>SPORT_READ_LINE</code> to terminate <code>READ</code> when a CR is encountered.</td>
</tr>
<tr>
<td>DL_TERM_LF</td>
<td>Causes <code>SPORT_READ_LINE</code> to terminate <code>READ</code> when a LF is encountered.</td>
</tr>
<tr>
<td>DL_TERM_CRLF</td>
<td>Causes <code>SPORT_READ_LINE</code> to terminate <code>READ</code> when CR+LF is encountered.</td>
</tr>
</tbody>
</table>

Passing a value of zero is the same as the following:

```
(DL_OUT_CR .OR. DL_TERM_CR .OR. DL_TOSS_CR .OR. DL_TOSS_LF)
```

This specifies to remove carriage returns and linefeeds on input, to follow output lines with a carriage return, and to return input lines when a carriage return is encountered.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** `SPORT_RELEASE`, *Using the Serial I/O Port Routines*, *Communications, Communications Functions*

**Example**

```fortran
USE DFLIB
INTEGER(4) iresult
iresult = SPORT_CONNECT( 2, 0 )
END
```

**SPORT_GET_HANDLE**

**Run-Time Function:** Returns the WIN32 handle associated with the communications port. This is the handle that was returned by the `CreateFile` Win32 API.

**Module:** USE DFLIB
**Syntax**

```plaintext
def iresult = SPORT_GET_HANDLE(port, handle)
```

*port* *(Input) Integer. The port number.*

*handle* *(Output) Integer(4). The WIN32 handle that was returned from CreatFile() on the serial port. This handle can be used with other WIN32 functions to set non-standard parameters for the communications port.*

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

```plaintext
USE DFLIB
INTEGER(4) iresult
INTEGER(4) handle
iresult = SPORT_GET_HANDLE(2, handle)
END
```

**SPORT_GET_STATE**

**Run-Time Function:** Returns the baud rate, parity, data bits setting, and stop bits setting of the communications port.

**Module:** USE DFLIB

**Syntax**

```plaintext
def iresult = SPORT_GET_STATE(port [, baud] [, parity] [, dbits] [, sbits])
```

*port* *(Input) Integer. The port number.*

*baud*
(Optional; output) Integer. The baud rate of the port.

\textit{parity}  
(Optional; output) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).

\textit{dbits}  
(Optional; output) Integer. The data bits for the port.

\textit{sbits}  
(Optional; output) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

\textbf{Results:}

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{SPORT\_SET\_STATE}, Using the Serial I/O Port Routines, Communications Functions

\textbf{Example}

```
USE DFLIB
INTEGER(4) iresult
INTEGER    baud
INTEGER    parity
INTEGER    dbits
INTEGER    sbits

iresult = SPORT\_GET\_STATE( 2, baud, parity, dbits, sbits )
END
```
### SPORT_GET_TIMEOUTS

*rx_int*
(Optional; output) Integer(4). The receive interval timeout value.

*tx_tot_mult*
(Optional; output) Integer(4). The transmit multiplier part of the timeout value.

*tx_tot_const*
(Optional; output) Integer(4). The transmit constant part of the timeout value.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** SPORT_SET_TIMEOUTS, Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

```fortran
USE DFLIB
INTEGER(4) iresult
INTEGER*4  rx_int
INTEGER*4  tx_tot_mult
INTEGER*4  tx_tot_const
    iresult =  SPORT_GET_TIMEOUTS( 2, rx_int, tx_tot_mult, tx_tot_const )
END
```

### SPORT_PEEK_DATA

**Run-Time Function:** Returns information about the availability of input data.

**Module:** USE DFLIB

**Syntax**

```fortran
    iresult = SPORT_PEEK_DATA (port [, present] [, count])
```

*port*

(Input) Integer. The port number.

*present*
(Optional; output) Integer. 1 if data is present, 0 if no data has been read.

\textit{count}
(Optional; output) Integer. The count of characters that will be returned by \texttt{SPORT_READ_DATA}.

\textbf{Results:}

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

\underline{Note:} CR and LF characters may not be returned depending on the mode specified in the \texttt{SPORT_CONNECT( )} call.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{SPORT_CONNECT}, \texttt{SPORT_READ_DATA}, \texttt{SPORT_PEEK_LINE}, Using the Serial I/O Port Routines, Communications, Communications Functions

\textbf{Example}

\begin{verbatim}
USE DFLIB
INTEGER(4) iresult
INTEGER    present
INTEGER    count

iresult =  SPORT_PEEK_DATA( 2, present, count )
END
\end{verbatim}

\textbf{SPORT_PEEK_LINE}

\textbf{Run-Time Function:} Returns information about the availability of input records.

\textbf{Module:} USE DFLIB

\textbf{Syntax}

\begin{verbatim}
iresult = \texttt{SPORT_PEEK_LINE}( \textit{port}, \textit{present}[, \textit{count}])
\end{verbatim}

\textit{port}
(Input) Integer. The port number.

\textit{present}
(Optional; output) Integer. 1 if data is present, 0 if no data has been read.
\( count \)
(Optional; output) Integer. The count of characters that will be returned by \texttt{SPORT_READ_DATA}.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the \texttt{SPORT_CONNECT}() call.

---

**Note:** CR and LF characters may not be returned depending on the mode specified in the \texttt{SPORT_CONNECT}() call.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** \texttt{SPORT_CONNECT}, \texttt{SPORT_READ_DATA}, \texttt{SPORT_PEEK_DATA}, Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

\begin{verbatim}
USE DFLIB
INTEGER(4) iresult
INTEGER    present
INTEGER    count

iresult = SPORT_PEEK_DATA( 2, present, count )
END
\end{verbatim}

**SPORT_PURGE**

**Run-Time Function:** Executes the Win32 API communications function \texttt{PurgeComm} on the specified port.

**Module:** USE DFLIB

**Syntax**

\[
\text{iresult} = \text{SPORT_PURGE} \ (\text{port}, \text{function})
\]

\( port \)
(Input) Integer. The port number.

\( function \)
(Input) Integer(4). The function for PurgeComm (see the WIN32 documentation).

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

USE DFWINTY
USE DFLIB
INTEGER(4) iresult
iresult = SPORT_PURGE( 2, (PURGE_TXABORT .or. PURGE_RXABORT) )
END

**SPORT_READ_DATA**

**Run-Time Function:** Reads available data from the specified port. This routine stalls until at least one character has been read.

**Module:** USE DFLIB

**Syntax**

\[
iresult = \text{SPORT_READ_DATA}(port, buffer [, count])
\]

*port*  
(Input) Integer. The port number.

*buffer*  
(Input) Character*(*)). The data that was read.

*count*  
(Optional; input) Integer. The count of bytes read.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.
Note: CR and LF characters may not be returned depending on the mode specified in the \texttt{SPORT \_CONNECT( )} call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: \texttt{SPORT \_CONNECT, SPORT \_PEEK \_DATA, SPORT \_READ \_LINE, SPORT \_WRITE \_DATA, Using the Serial I/O Port Routines, Communications, Communications Functions}

Example

\begin{verbatim}
USE DFLIB
INTEGER(4)  iresult
INTEGER       count
CHARACTER*1024  rbuff

iresult = SPORT\_READ\_DATA( 2, rbuff, count )
END
\end{verbatim}

\textbf{SPORT\_READ\_LINE}

\textbf{Run-Time Function:} Reads a record from the specified port. This routine stalls until at least one record has been read.

\textbf{Module: USE DFLIB}

\textbf{Syntax}

\begin{verbatim}
iresult = SPORT\_READ\_LINE ( port, buffer [, count] )
\end{verbatim}

\textit{port} \\
(Input) Integer. The port number.

\textit{buffer} \\
(Input) Character*(*)\textdagger. The data that was read.

\textit{count} \\
(Optional; input) Integer. The count of bytes read.

\textbf{Results:}

The result type is \texttt{INTEGER(4)}. The result is zero if successful; otherwise, a \texttt{WIN32} error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the \texttt{SPORT \_CONNECT( )} call.
Note: CR and LF characters may not be returned depending on the mode specified in the `SPORT_CONNECT` call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: `SPORT_CONNECT`, `SPORT_PEEK_LINE`, `SPORT_READ_DATA`, `SPORT_WRITE_LINE`, Using the Serial I/O Port Routines, Communications, Communications Functions

Example

```fortran
USE DFLIB
INTEGER(4)      iresult
INTEGER         count
CHARACTER*1024   rbuff

iresult = SPORT_READ_LINE( 2, rbuff, count )
END
```

**SPORT_RELEASE**

Run-Time Function: Releases a serial port that was previously connected to (by using `SPORT_CONNECT`).

Module: USE DFLIB

Syntax

```
ireult = SPORT_RELEASE (port)
```

- `port`  
  (Input) Integer. The port number.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: `SPORT_CONNECT`, Using the Serial I/O Port Routines, Communications, Communications Functions
SPORT_SET_STATE

Run-Time Function: Sets the baud rate, parity, data bits setting, and stop bits setting of the communications port.

Module: USE DFLIB

Syntax

iresult = SPORT_SET_STATE (port [, baud] [, parity] [, dbits] [, sbits])

port
(Input) Integer. The port number.

baud
(Optional; input) Integer. The baud rate of the port.

parity
(Optional; input) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).

dbits
(Optional; input) Integer. The data bits for the port.

sbits
(Optional; input) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

Note: This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB
SPORT_SET_STATE

**See Also:** SPORT_CANCEL_IO, SPORT_GET_STATE, Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

```fortran
USE DFLIB
INTEGER(4) iresult
iresult = SPORT_SET_STATE( 2, 9600, 0, 7, 1 )
END
```

### SPORT_SET_TIMEOUTS

**Run-Time Function:** Sets the user selectable timeouts for the serial port.

**Module:** USE DFLIB

**Syntax**

```fortran
iresult = SPORT_SET_TIMEOUTS (port [, rx_int] [, tx_tot_mult] [, tx_tot_const])
```

- **port**
  (Input) Integer. The port number.

- **rx_int**
  (Optional; input) Integer(4). The receive interval timeout value.

- **tx_tot_mult**
  (Optional; input) Integer(4). The transmit multiplier part of the timeout value.

- **tx_tot_const**
  (Optional; input) Integer(4). The transmit constant part of the timeout value.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Note:** This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORTANCEL_IO before port parameters can be changed.

**Compatibility**
SPORT_SET_TIMEOUTS

See Also: SPORTCANCEL_IO, SPORT_GET_TIMEOUTS, Using the Serial I/O Port Routines, Communications, Communications Functions

Example

USE DFLIB
INTEGER(4) iresult
iresult = SPORT_SET_TIMEOUTS( 2, 100, 0, 1000 )
END

SPORT_SHOW_STATE

Run-Time Function: Displays the state of a port to standard output.

Module: USE DFLIB

Syntax

iresult = SPORT_SHOW_STATE (port, level)

port
(Input) Integer. The port number.

level
(Input) Integer. Controls the level of detail displayed as follows:

0 Basic one line display
1 Basic information
2 Add modem signal control flow information
3 Add XON/XOFF information
4 Add event character information
11 Add timeout information
901 Add debug information

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.
**Note:** This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call **SPORT_CANCEL_IO** before port parameters can be changed.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** **SPORT_CANCEL_IO**, **Using the Serial I/O Port Routines**, **Communications**, **Communications Functions**

**Example**

```fortran
USE DFLIB
INTEGER(4) iresult
iresult = SPORT_SHOW_STATE( 2, 0 )
END
```

**SPORT_SPECIAL_FUNC**

**Run-Time Function:** Executes the Win32 API communications function **EscapeCommFunction** on the specified port.

**Module:** USE DFLIB

**Syntax**

```fortran
iresult = SPORT_SPECIAL_FUNC (port, function)
```

- **port**
  (Input) Integer. The port number.

- **function**
  (Input) Integer(4). The function to perform.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** **Using the Serial I/O Port Routines**, **Communications**, **Communications Functions**

**Example**
USE DFLIB
INTEGER(4) iresult
iresult = SPORT_SPECIAL_FUNC( 2, ? )
END

SPORT_WRITE_DATA

Run-Time Function: Outputs data to the specified port.

Module: USE DFLIB

Syntax

iresult = SPORT_WRITE_DATA( port, data [, count] )

port
(Input) Integer. The port number.

data
(Input) Character*(*). The data to be output.

count
(Optional; input) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

Results:

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also: SPORT_WRITE_LINE, SPORT_READ_DATA, Using the Serial I/O Port Routines, Communications, Communications Functions

Example

USE DFLIB
INTEGER(4) iresult
iresult = SPORT_WRITE_DATA( 2, 'ATZ'//CHAR(13), 0 )
END

SPORT_WRITE_LINE
**Run-Time Function:** Outputs data, followed by a record terminator, to the specified port.

**Module:** USE DFLIB

**Syntax**

```fortran
  ireal = SPORT_WRITE_LINE( port, data [, count] )
```

- **port** (Input) Integer. The port number.
- **data** (Input) Character*(*) — The data to be output.
- **count** (Optional; Input) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

**Results:**

The result type is INTEGER(4). The result is zero if successful; otherwise, a WIN32 error value.

After the data is output, a line terminator character is added based on the mode used during the **SPORT_CONNECT**() call.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** SPORT_CONNECT, SPORT_WRITE_DATA, SPORT_READ_DATA, Using the Serial I/O Port Routines, Communications, Communications Functions

**Example**

```fortran
USE DFLIB
INTEGER(4) ireal
ireal = SPORT_WRITE_LINE( 2, 'ATZ', 0 )
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**

\[
\text{result} = \text{SPREAD}\left(\text{source, dim, ncopies}\right)
\]

*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* \(\leq\) 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

\[
\begin{bmatrix}
3 & 4 & 5 \\
3 & 4 & 5 \\
3 & 4 & 5 \\
3 & 4 & 5 \\
\end{bmatrix}
\]
SPREAD (B, DIM=2, NCOPIES=NC) produces the array

\[
\begin{bmatrix}
3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5
\end{bmatrix}.
\]

The following shows another example:

```fortran
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**

\[
\text{result} = \text{SQRT} \ (x)
\]

\(x\)

(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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQRT</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DSQRT</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QSQRT(^1)</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
<tr>
<td>CSQRT(^2)</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
</tr>
<tr>
<td>CDSQRT(^3)</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
</tr>
</tbody>
</table>
### 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:

```fortran
! 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 x86 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**

\[
\text{result} = \text{STAT} \ (\text{name}, \text{statb})
\]

\textit{name}

(Input) Character*(*). Name of the file to examine.

\textit{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 \textit{statb} contain the following values:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>statb(1)</td>
<td>Device file resides on</td>
<td>Always 0</td>
</tr>
<tr>
<td>statb(2)</td>
<td>File Inode number</td>
<td>Always 0</td>
</tr>
<tr>
<td>statb(3)</td>
<td>Access mode of the file</td>
<td>(See following table)</td>
</tr>
<tr>
<td>statb(4)</td>
<td>Number of hard links</td>
<td>Always 1</td>
</tr>
<tr>
<td>statb(5)</td>
<td>User ID of owner</td>
<td>Always 1</td>
</tr>
<tr>
<td>statb(6)</td>
<td>Group ID of owner</td>
<td>Always 1</td>
</tr>
<tr>
<td>statb(7)</td>
<td>Raw device file resides on</td>
<td>Always 0</td>
</tr>
<tr>
<td>statb(8)</td>
<td>Size of the file in bytes</td>
<td></td>
</tr>
<tr>
<td>statb(9)</td>
<td>Time when the file was last accessed</td>
<td>(Only available on non-FAT file systems; undefined on FAT systems)</td>
</tr>
<tr>
<td>statb(10)</td>
<td>Time when the file was last modified</td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Constant</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IFMT</td>
<td>O'0170000'</td>
<td>Type of File</td>
<td></td>
</tr>
<tr>
<td>S_IFDIR</td>
<td>O'0040000'</td>
<td>Directory</td>
<td></td>
</tr>
<tr>
<td>S_IFCHR</td>
<td>O'0020000'</td>
<td>Character Special</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IFBLK</td>
<td>O'0060000'</td>
<td>Block Special</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IFREG</td>
<td>O'0100000'</td>
<td>Regular</td>
<td></td>
</tr>
<tr>
<td>S_IFLNK</td>
<td>O'0120000'</td>
<td>Symbolic Link</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IFSOCK</td>
<td>O'0140000'</td>
<td>Socket</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISUID</td>
<td>O'0004000'</td>
<td>Set User ID on Execution</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISGID</td>
<td>O'0002000'</td>
<td>Set Group ID on Execution</td>
<td>Never set</td>
</tr>
<tr>
<td>S_ISVTX</td>
<td>O'0001000'</td>
<td>Save Swapped Text</td>
<td>Never set</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>O'0000700'</td>
<td>Owner's File Permissions</td>
<td></td>
</tr>
<tr>
<td>S_IRUSR, S_IREAD</td>
<td>O'0000400'</td>
<td>Owner Read Permission</td>
<td>Always true</td>
</tr>
</tbody>
</table>
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 STANDALONE 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**

\[
\text{fun} \hspace{1em} ([\text{d-arg} [, \text{d-arg} ] ...]) = \text{expr}
\]

- **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:

\[
\text{fun} \hspace{1em} ([\text{a-arg} [, \text{a-arg} ] ...])
\]

- **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:

```plaintext
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:

```plaintext
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
```
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:

```fortran
REAL COMP, C, D, E
COMP (C,D,E,3.) = (C + D - E)/3.
```

The following shows another example:

```fortran
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.

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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>STATIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy argument</td>
<td>No</td>
</tr>
<tr>
<td>Automatic object</td>
<td>No</td>
</tr>
<tr>
<td>Common block item</td>
<td>Yes</td>
</tr>
<tr>
<td>Use-associated item</td>
<td>No</td>
</tr>
</tbody>
</table>
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
!  ```
**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 95/90 and FORTRAN 77 limit digits to five.)

**Effect on Windows NT (including Windows 2000) and Windows 9* 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 Tru64 UNIX and Linux 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

General Compiler Directive: STRICT disables language features not found in the language standard specified on the command line (Fortran 95 or Fortran 90). NOSTRICT (the default) enables these features.

Syntax
cDEC$ STRICT
cDEC$ NOSTRICT

C
Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 90.

The STRICT and NOSTRICT directives can appear only 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, /stand

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
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:

  ```plaintext
  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**

**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:

```plaintext
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 (.)

```plaintext
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`.

```plaintext
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
```
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**

```
(byte offset)

0   field DAY of field APP_DATE
1   field MONTH of field APP_DATE
2   field YEAR of field APP_DATE
3
4   field HOUR of field APP_TIME(1)
5   field MINUTE of field APP_TIME(1)
6   field HOUR of field APP_TIME(2)
7   field MINUTE of field APP_TIME(2)
8   field APP_MEMO(1)

...                               

28   field APP_MEMO(2)

...                               

48   field APP_MEMO(3)

...                               

68   field APP_MEMO(4)

...                               

88   field APP_FLAG

89   ZK-1848-GE
```

**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

\[
[prefix] \text{SUBROUTINE} \ name \ [(\ [d-arg-list]\ )]
\]

prefix
(Optional) Is one of the following:

\[
type \ [\text{keyword}]
\]

\[
\text{keyword} \ [\text{type}]
\]

type
Is a data type specifier.

keyword
Is one of the following:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECURSIVE</td>
<td>Permits direct recursion to occur.</td>
</tr>
<tr>
<td>PURE</td>
<td>Asserts that the procedure has no side effects.</td>
</tr>
<tr>
<td>ELEMENTAL</td>
<td>Restricted form of pure procedure that acts on one array element at a time.</td>
</tr>
</tbody>
</table>

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:

```fortran
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:

```fortran
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:

```fortran
SUBROUTINE GetNum (num, unit)
  INTEGER num, unit
  10  READ (unit, '(I10)', ERR = 10) num
END
```

**SUBTITLE**

**General Compiler Directive:** Specifies a string for the subtitle field of a listing header.

**Syntax**

```fortran
cDEC$ 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**

```plaintext
!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**

```plaintext
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 \texttt{array}.

The result is a scalar if \texttt{dim} is omitted or \texttt{array} has rank one.

The following rules apply if \texttt{dim} is omitted:

- If \texttt{SUM(array)} is specified, the result is the sum of all elements of \texttt{array}. If \texttt{array} has size zero, the result is zero.

- If \texttt{SUM(array, MASK=mask)} is specified, the result is the sum of all elements of \texttt{array} corresponding to true elements of \texttt{mask}. If \texttt{array} has size zero, or every element of \texttt{mask} has the value .FALSE., the result is zero.

The following rules apply if \texttt{dim} is specified:

- If \texttt{array} has rank one, the value is the same as \texttt{SUM(array [,MASK=mask])}.

- An array result has a rank that is one less than \texttt{array}, and shape \((d_1, d_2, \ldots, d_{\text{dim}-1}, d_{\text{dim}+1}, \ldots, d_n)\), where \((d_1, d_2, \ldots, d_n)\) is the shape of \texttt{array}.

- The value of element \((s_1, s_2, \ldots, s_{\text{dim}-1}, s_{\text{dim}+1}, \ldots, s_n)\) of \texttt{SUM(array, dim [,mask])} is equal to \texttt{SUM(array (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n) [,MASK = mask (s_1, s_2, \ldots, s_{\text{dim}-1}, :, s_{\text{dim}+1}, \ldots, s_n)])}.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** PRODUCT

**Examples**

\texttt{SUM ((/2, 3, 4/))} returns the value 9 (sum of 2 + 3 + 4). \texttt{SUM ((/2, 3, 4/), DIM=1)} returns the same result.

\texttt{SUM (B, MASK=B .LT. 0.0)} returns the arithmetic sum of the negative elements of \texttt{B}.

\texttt{C} is the array

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

\texttt{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:

```fortran
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**

```fortran
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.

On Windows NT (including Windows 2000) systems, the calling process waits until the command terminates. On Windows 9x systems, the calling process does not currently wait in all cases; however, this may change in future implementations. To insure compatibility and consistent behavior, an image can
be invoked directly by using the WIN32 API CreateProcess ( ) in your Fortran code.

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, errno
I = SYSTEM("dir > file.lst")
If (I .eq. -1) then
   errno = ierrno( )
   print *, 'Error ', errno
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:

```fortran
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**

```fortran
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 **SYSTEM** function lets you pass operating-system commands as well as programs. **SYSTEM** refers to the COMSPEC and PATH environment variables that locate the command interpreter file (usually named COMMAND.COM).

On Windows NT (including Windows 2000) systems, the calling process waits until the command terminates. On Windows 9x systems, the calling process does not currently wait in all cases; however, this may change in future implementations. To insure compatibility and consistent behavior, an image can be invoked directly by using the WIN32 API **CreateProcess** ( ) in your Fortran code.

If the function fails, call **GETLASTERROR** 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 **SYSTEM** 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.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAN</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DTAN</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QTAN ¹</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

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 \( x \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAND</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DTAND</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QTAND {1}</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

{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**

\[
\text{result} = \text{TANH} \ (x)
\]

\( x \)  
(Input) Must be of type real.

**Results:**

The result type is the same as \( x \).

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANH</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DTANH</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
</tr>
<tr>
<td>QTANH {1}</td>
<td>REAL(16)</td>
<td>REAL(16)</td>
</tr>
</tbody>
</table>

{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:

\[ \text{type, } [\text{att-}ls,] \text{ TARGET } [, \text{att-}ls] :: \text{object } [(\text{a-spec})] [, \text{object } [(\text{a-spec})]] ... \]

Statement:

\[ \text{TARGET } ::\] object [(\text{a-spec})] [, object [(\text{a-spec})]] ... \]

\text{type}
Is a data type specifier.

\text{att-}ls
Is an optional list of attribute specifiers.

\text{object}
Is the name of the object. The object must not be declared with the PARAMETER attribute.

\text{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
...  
```

TASKCOMMON (TU*X only)

Compaq Fortran Parallel Compiler Directive: Makes named common blocks private to a thread but global within the thread. This directive is the same as the OpenMP Fortran API THREADPRIVATE directive except that slashes (/ /) do not have to be used to delimit named common blocks.

See Also: Parallel Directives for Tru64 UNIX Systems, Compaq Fortran Parallel Compiler Directives (TU*X only), OpenMP Fortran API Compiler Directives (TU*X only)

THREADPRIVATE (TU*X only)
OpenMP Parallel Compiler Directive: Specifies named common blocks to be private (local) to a thread; they are global within the thread.

Syntax

```c
$OMP THREADPRIVATE ( /cb/ [, /cb/] ... )
```

- **c**
  - Is one of the following: C (or c), !, or * (see Syntax Rules for Parallel Directives).

- **cb**
  - Is the name of the common block you want made private to a thread. Only named common blocks can be made thread private. Note that the slashes ( / ) are required.

Rules and Behavior

Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads.

During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block. On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified in the PARALLEL directive.

When a common block (which is initialized using DATA statements) appears in a THREADPRIVATE directive, each thread copy is initialized once prior to its first use. For subsequent parallel regions, data in THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

A THREADPRIVATE common block or its constituent variables can appear only in a COPYIN clause. They are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

See Also: Parallel Directives for Tru64 UNIX Systems, OpenMP Fortran API Compiler Directives (TU*X only), Compaq Fortran Parallel Compiler Directives (TU*X only)

Examples

In the following example, the common blocks BLK1 and FIELDS are specified as thread private:
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**

```plaintext
result = TIME()
```

**Subroutine Syntax**

```plaintext
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](#)
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 \( b^{e_{\text{min}}-1} \). Parameters \( b \) and \( e_{\text{min}} \) 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**

General Compiler Directive: Specifies a string for the title field of a listing header.

Syntax

```
cDEC$ 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
```

**TRACEBACKQQ**

**Run-Time Subroutine:** Provides traceback information. Uses the Visual Fortran run-time library traceback facility to generate a stack trace showing the program call stack as it appeared at the time of the call to `TRACEBACKQQ( )`.

**Module:** USE DFLIB

**Syntax**

```fortran
CALL TRACEBACKQQ ([string] [, user_exit_code] [, status] [, eptr])
```

*string*  
(Optional; input) CHARACTER*(*)

A message string to precede the traceback output. It is recommended that the string be no more than 80 characters (one line) since that length appears better on output. However, this limit is not a restriction and it is not enforced. The string is output exactly as specified; no formatting or interpretation is done.

If this argument is omitted, no header message string is produced.
user_exit_code
(Optional; input) INTEGER(4). An exit code. Two values are predefined:

- A value of -1 causes the run-time system to return execution to the caller after producing traceback.
- A value of zero (the default) causes the application to abort execution.

Any other specified value causes the application to abort execution and return the specified value to the operating system.

status
(Optional; input) INTEGER(4). A status value. If specified, the run-time system returns the status value to the caller indicating that the traceback process was successful. The default is not to return status.

Note that a returned status value is only an indication that the "attempt" to trace the call stack was completed successfully, not that it produced a useful result.

You can include the file iosdef.for in your program to obtain symbolic definitions for the possible return values. A return value of FOR$IOS_SUCCESS (0) indicates success.

eptr
(Optional; input) Cray pointer. It is required if calling from a user-specified exception filter. If omitted, the default in null.

To trace the stack after an exception has occurred, the runtime support needs access to the exception information supplied to the filter by the operating system.

The eptr argument is a pointer to T_EXCEPTION_POINTERS, returned by the WIN32 API GetExceptionInformation(), which is usually passed to a C try/except filter function. This argument must be null if you are not passing a valid pointer to T_EXCEPTION_POINTERS. For more information, see Obtaining Traceback Information with TRACEBACKQQ in the Programmer's Guide.

The TRACEBACKQQ routine provides a standard way for an application to initiate a stack trace. It can be used to report application detected errors, debugging, and so forth. It uses the stack trace support in the Visual Fortran run-time library, and produces the same output that the run-time library produces for unhandled errors and exceptions.

The error message string normally included by the run-time system is replaced with the user-supplied message text, or omitted if no string is specified. Traceback output is directed to the target destination appropriate for the
application type, just as it is when traceback is initiated internally by the run-time system.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** GETEXCEPTIONPTRSQQ, Obtaining Traceback Information with TRACEBACKQQ, Using Traceback Information, Run-Time Message Display and Format

**Examples**

Consider the following:

```fortran
CALL TRACEBACKQQ()
```

This example generates a traceback report with no leading header message, from wherever the call site is, and aborts execution.

The following is another example:

```fortran
CALL TRACEBACKQQ("My application message string")
```

This example generates a traceback report with the user-supplied string as the header, and aborts execution.

The following is another example:

```fortran
CALL TRACEBACKQQ(STRING="Bad value for TEMP",USER_EXIT_CODE=123)
```

This example generates a traceback report with the user-supplied string as the header, and aborts execution, returning a status code of 123 to the operating system.

The following is another example:

```fortran
...
INTEGER(4) RTN_STS
INCLUDE 'IOSDEF.FOR'
...
CALL TRACEBACKQQ(USER_EXIT_CODE=-1,STATUS=RTN_STS)
IF (RTN_STS .EQ. FOR$IOS_SUCCESS) THEN
  PRINT *, 'TRACEBACK WAS SUCCESSFUL'
END IF
...
```

This example generates a traceback report with no header string, and returns to the caller to continue execution of the application. If the traceback process succeeds, a status will be returned in variable RTN_STS.
For more examples, including one showing a Cray pointer, see Obtaining Traceback Information with TRACEBACKQQ in the Programmer's Guide.

**TRAILZ**

**Elemental Intrinsic Function (Generic):** Returns the number of trailing zero bits in an integer.

**Syntax**

\[
\text{result} = \text{TRAILZ}(i)
\]

\(i\)

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:

```fortran
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**

\[
\text{result} = \text{TRANSFER}(\text{source}, \text{mold} [, \text{size}])
\]

\(\text{source}\)

(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:

```fortran
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**

```fortran
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

\[
\begin{bmatrix}
2 & 5 & 8 \\
3 & 6 & 9 \\
4 & 7 & 1
\end{bmatrix}
\]

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
```

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:

```fortran
! 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:

```fortran
 type [ [, att ] ... :: ] v [/c-list/] [, v [/c-list/] ] ... 
```

```fortran
 type 
```

Is one of the following data type specifiers:

```fortran
 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:

<table>
<thead>
<tr>
<th>ATTRIBUTION</th>
<th>INTRINSIC</th>
<th>PUBLIC 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>OPTIONAL</td>
<td>SAVE</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>PARAMETER</td>
<td>STATIC</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>POINTER</td>
<td>TARGET</td>
</tr>
<tr>
<td>INTENT</td>
<td>PRIVATE 1</td>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

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.

\textit{c-list}

Is a list of constants, as in a \textbf{DATA} statement. If v is the name of a constant or an initialization expression, the \textit{c-list} cannot be present.

The \textit{c-list} cannot specify more than one value unless it initializes an array. When initializing an array, the \textit{c-list} must contain a value for every element in the array.

\textbf{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 \textbf{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 \textit{att} appears, \textit{c-list} cannot be specified; for example:

\begin{verbatim}
    INTEGER I /2/            ! Valid
    INTEGER, SAVE :: I /2/   ! Invalid
\end{verbatim}

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Compatible with:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATABLE</td>
<td>AUTOMATIC, DIMENSION, PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>ALLOCATABLE, DIMENSION, POINTER, TARGET, VOLATILE</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>ALLOCATABLE, AUTOMATIC, INTENT, OPTIONAL, PARAMETER, TARGET, VOLATILE</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>OPTIONAL, PRIVATE, PUBLIC</td>
</tr>
<tr>
<td>INTENT</td>
<td>DIMENSION, OPTIONAL, TARGET, VOLATILE</td>
</tr>
<tr>
<td>INTRINSIC</td>
<td>PRIVATE, PUBLIC</td>
</tr>
<tr>
<td>OPTIONAL</td>
<td>DIMENSION, EXTERNAL, INTENT, POINTER, TARGET, VOLATILE</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>DIMENSION, PRIVATE, PUBLIC</td>
</tr>
</tbody>
</table>
### Type Declarations

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Automatic, Dimension ¹, Optional, Private, Public, Save, Static, Volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>Allocatable, Dimension, External, Intrinsic, Parameter, Pointer, Save, Static, Target, Volatile</td>
</tr>
<tr>
<td>Public</td>
<td>Allocatable, Dimension, External, Intrinsic, Parameter, Pointer, Save, Static, Target, Volatile</td>
</tr>
<tr>
<td>Save</td>
<td>Allocatable, Dimension, Pointer, Private, Public, Static, Target, Volatile</td>
</tr>
<tr>
<td>Static</td>
<td>Allocatable, Dimension, Pointer, Private, Public, Save, Target, Volatile</td>
</tr>
<tr>
<td>Target</td>
<td>Allocatable, Automatic, Dimension, Intent, Optional, Private, Public, Save, Static, Volatile</td>
</tr>
<tr>
<td>Volatile</td>
<td>Allocatable, Automatic, Dimension, Intent, Optional, Pointer, Private, Public, Save, Static, Target</td>
</tr>
</tbody>
</table>

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:

```fortran
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 (0.0D0)) a1
REAL (KIND = 2) b
COMPLEX (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] [, kind])

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.

kind
(Optional; input) Must be a scalar integer initialization expression.

Results:

The result type is integer. If kind is present, the kind parameter of the result is that specified by kind; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If dim is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of array. Each element in the result corresponds to a dimension of array.

If array is an array section or an array expression that is not a whole array or
array structure component, \texttt{UBOUND(array, dim)} has a value equal to the number of elements in the given dimension.

If \textit{array} is a whole array or array structure component, \texttt{UBOUND(array, dim)} has a value equal to the upper bound for subscript \textit{dim} of \textit{array} (if \textit{dim} is nonzero). If \textit{dim} has size zero, the corresponding element of the result has the value zero.

The setting of compiler options that specify integer size can affect the result of this function.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} \texttt{LBOUND}

\textbf{Examples}

Consider the following:

\begin{verbatim}
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
\end{verbatim}

\texttt{UBOUND (ARRAY_A)} is (3, 8). \texttt{UBOUND (ARRAY_A, DIM=2)} is 8.

\texttt{UBOUND (ARRAY_B)} is (8, 20). \texttt{UBOUND (ARRAY_B (5:8, :) )} is (4,24) because the number of elements is significant for array section arguments.

The following shows another example:

\begin{verbatim}
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{verbatim}

\textbf{UNION...END UNION}

\textbf{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.

\textbf{Syntax}

\begin{verbatim}
UNION
\end{verbatim}
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:

```plaintext
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**

![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:

```plaintext
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
```
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*

*i-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

\[
\begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 1 \\
1 & 0 & 0
\end{bmatrix},
\]

P is the array (2, 3, 4, 5), and Q is the array

\[
\begin{bmatrix}
T & F & F \\
F & T & F \\
T & T & F
\end{bmatrix}.
\]

UNPACK (P, MASK=Q, FIELD=N) produces the result

\[
\begin{bmatrix}
2 & 0 & 1 \\
1 & 4 & 1 \\
3 & 5 & 0
\end{bmatrix}.
\]

UNPACK (P, MASK=Q, FIELD=1) produces the result

\[
\begin{bmatrix}
2 & 1 & 1 \\
1 & 4 & 1
\end{bmatrix}.
The following shows another example:

```plaintext
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**

```plaintext
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 (FILEINFO) 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$LBUTTONDLCLK - Left mouse button double-click
- MOUSE$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 UNREGISTERMOUSEEEVENT, QuickWin no longer calls the callback routine specified earlier for the window when mouse events occur. Calling UNREGISTERMOUSEEEVENT when no callback routine is registered for the window has no effect.

Compatibility

QUICKWIN GRAPHICS LIB

See Also: Using QuickWin, REGISTERMOUSEEEVENT, WAITONMOUSEEEVENT

UNROLL

General Compiler Directive: Tells the compiler's optimizer how many times to unroll a DO loop. It can only be applied to iterative DO loops.

Syntax

```
cDEC$ UNROLL [(n)]
```

- c
  Is one of the following: C (or c), !, or *. (See Syntax Rules for General Directives.)

- n
  Is an integer constant. The range of n is 0 through 255.

Rules and Behavior

The UNROLL directive must precede the DO statement for each DO loop it affects. No source code lines, other than the following, can be placed between the UNROLL directive statement and the DO statement:

- An IVDEP directive
- A PARALLEL DO directive (TU*X only)
If \( n \) is specified, the optimizer unrolls the loop \( n \) times. If \( n \) is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The **UNROLL** directive overrides any setting of loop unrolling from the command line.

**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**See Also:** [General Compiler Directives](#)

**USE**

**Statement:** Gives a program unit accessibility to public entities in a module.

**Syntax**

```plaintext
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:
    ```plaintext
    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:
    ```plaintext
    [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-lists and only-lists are interpreted as a single, concatenated rename-list.
- If all the USE statements have ONLY options, all the only-lists are interpreted as a single, concatenated only-list. Only those entities named in one or more of the only-lists 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:

```plaintext
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**

\[
\text{result} = \%\text{VAL} (a)
\]

\(a\)

(Input) An expression, record name, procedure name, array, character array section, or array element.

The argument is passed as follows:

- On x86 processors, as a 32-bit immediate value. If the argument is integer (or logical) and shorter than 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 than 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 Compaq Fortran defaults for argument passing, and the allowed uses of \%VAL:

<table>
<thead>
<tr>
<th>Actual Argument Data Type</th>
<th>Default</th>
<th>%VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressions:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>REF</td>
<td>Yes1</td>
</tr>
</tbody>
</table>
The %VAL, %REF, and %DESCR functions override related cDEC$ ATTRIBUTE

<table>
<thead>
<tr>
<th>Type</th>
<th>Reference</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>REF</td>
<td>Yes¹</td>
</tr>
<tr>
<td>REAL(4)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>REAL(8)</td>
<td>REF</td>
<td>Yes²</td>
</tr>
<tr>
<td>REAL(16)</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>COMPLEX(4)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>COMPLEX(8)</td>
<td>REF</td>
<td>Yes</td>
</tr>
<tr>
<td>COMPLEX(16)</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Character</td>
<td>See table note⁴</td>
<td>No</td>
</tr>
<tr>
<td>Hollerith</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Aggregate⁵</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Derived</td>
<td>REF</td>
<td>No</td>
</tr>
</tbody>
</table>

**Array Name:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Reference</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Character</td>
<td>See table note⁴</td>
<td>No</td>
</tr>
<tr>
<td>Aggregate⁵</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Derived</td>
<td>REF</td>
<td>No</td>
</tr>
</tbody>
</table>

**Procedure Name:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Reference</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>REF</td>
<td>No</td>
</tr>
<tr>
<td>Character</td>
<td>See table note⁴</td>
<td>No</td>
</tr>
</tbody>
</table>

¹ If a logical or integer value occupies less than 64 (Alpha systems) or 32 (x86 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, a character argument is passed by address and hidden length.
⁵ In Compaq Fortran record structures
settings.

See Also: **CALL, %REF, %DESCR**

**Example**

```fortran
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**

```fortran
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**
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:

```fortran
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:**

```fortran
type, [ att-ls, ] VOLATILE [, att-ls ] :: object [, object] ...
```

**Statement:**

```fortran
VOLATILE object [, object] ...
```

*type*

Is a data type specifier.
att-ls
Is an optional list of attribute specifiers.

object
Is the name of an object, or the name of a common block enclosed in slashes.

Rules and Behavior

A variable or COMMON block must be declared VOlATile if it can be read or written in a way that is not visible to the compiler. For example:

- If an operating system feature is used to place a variable in shared memory (so that it can be accessed by other programs), the variable must be declared VOLATILE.
- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.

If an array is declared VOLATILE, each element in the array becomes volatile. If a common block is declared VOLATILE, each variable in the common block becomes volatile.

If an object of derived type is declared VOLATILE, its components become volatile.

If a pointer is declared VOLATILE, the pointer itself becomes volatile.

A VOLATILE statement cannot specify the following:

- 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 \textbf{VOLATILE} statement:

\begin{verbatim}
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)
\end{verbatim}

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.

\section*{WAITONMOUSEEVENT}

\textbf{QuickWin Function:} Waits for the specified mouse input from the user.

\textbf{Module: USE DFLIB}

\textbf{Syntax}

\begin{verbatim}
result = WAITONMOUSEEVENT (mouseevents, keystate, x, y)
\end{verbatim}

\textit{mouseevents}

(Input) INTEGER(4). One or more mouse events that must occur before the function returns. Symbolic constants for the possible mouse events are:

- \texttt{MOUSE$LBUTTONDOWN} - Left mouse button down
- \texttt{MOUSE$LBUTTONUP} - Left mouse button up
- \texttt{MOUSE$LBUTTONDBLCLK} - Left mouse button double-click
- \texttt{MOUSE$RBUTTONDOWN} - Right mouse button down
- \texttt{MOUSE$RBUTTONUP} - Right mouse button up
- \texttt{MOUSE$RBUTTONDBLCLK} - Right mouse button double-click
- \texttt{MOUSE$MOVE} - Mouse moved

\textit{keystate}

(Output) INTEGER(4). Bitwise inclusive OR of the state of the mouse during the event. The value returned in \textit{keystate} can be any or all of the following symbolic constants:

- \texttt{MOUSE$KS_LBUTTON} - Left mouse button down during event
- \texttt{MOUSE$KS_RBUTTON} - Right mouse button down during event
- \texttt{MOUSE$KS_SHIFT} - SHIFT key held down during event
- \texttt{MOUSE$KS_CONTROL} - CONTROL key held down during event
WAITONMOUSEEVENT

\[ x \]
(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 Construct:** 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** \((\text{mask-expr1})\) \text{assign-stmt}

**Contract:**

\[
[\text{name}]: \text{WHERE} \ (\text{mask-expr1}) \\
\ [\text{where-body-stmt}] ... \\
\ [\text{ELSEWHERE} \ (\text{mask-expr2}) \ [\text{name}] \\
\ [\text{where-body-stmt}] ...] \\
\ [\text{ELSEWHERE} \ [\text{name}] \\
\ [\text{where-body-stmt}] ...] \\
\text{END \ WHERE} \ [\text{name}]
\]

\text{mask-expr1, mask-expr2}

Are logical array expressions (called mask expressions).

\text{assign-stmt}

Is an assignment statement of the form: array variable = array expression.

\text{name}

Is the name of the \text{WHERE} construct.

\text{where-body-stmt}

Is one of the following:

- An \text{assign-stmt}

  The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.

- A \text{WHERE} statement or construct

**Rules and Behavior**

If a construct \text{name} is specified in a \text{WHERE} statement, the same name must appear in the corresponding \text{END \ WHERE} statement. The same construct name can optionally appear in any \text{ELSEWHERE} statement in the construct. (\text{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 \text{WHERE} statement (or the first line of the \text{WHERE} construct) can be labeled as a branch target statement.

The following shows an example using a \text{WHERE} statement:
INTEGER A, B, C
DIMENSION A(5), B(5), C(5)
DATA A /0,1,1,1,0/
DATA B /10,11,12,13,14/
C = -1
WHERE(A .NE. 0) C = B / A

The resulting array C contains: -1,11,12,13, and -1.

The assignment statement is only executed for those elements where the mask is true. Think of the mask expression as being evaluated first into a logical array that has the value true for those elements where A is positive. This array of trues and falses is applied to the arrays A, B and C in the assignment statement. The right side is only evaluated for elements for which the mask is true; assignment on the left side is only performed for those elements for which the mask is true. The elements for which the mask is false do not get assigned a value.

In a WHERE construct, the mask expression is evaluated first and only once. Every assignment statement following the WHERE is executed as if it were a WHERE statement with "mask-expr1" and every assignment statement following the ELSEWHERE is executed as if it were a WHERE statement with "NOT. mask-expr1". If ELSEWHERE specifies "mask-expr2", it is executed as "(.NOT. mask-expr1) .AND. mask-expr2" during the processing of the ELSEWHERE statement.

You should be careful if the statements have side effects, or modify each other or the mask expression.

The following is an example of the WHERE construct:

```
DIMENSION PRESSURE(1000), TEMP(1000), PRECIPITATION(1000)
WHERE(PRESSURE .GE. 1.0)
   PRESSURE = PRESSURE + 1.0
   TEMP = TEMP - 10.0
ELSEWHERE
   PRECIPITATION = .TRUE.
ENDWHERE
```

The mask is applied to the arguments of functions on the right side of the assignment if they are considered to be elemental functions. Only elemental intrinsics are considered elemental functions. Transformational intrinsics, inquiry intrinsics, and functions or operations defined in the subprogram are considered to be nonelemental functions.

Consider the following example using LOG, an elemental function:

```
WHERE(A .GT. 0)  B = LOG(A)
```
The mask is applied to A, and \texttt{LOG} is executed only for the positive values of A. The result of the \texttt{LOG} is assigned to those elements of B where the mask is true.

Consider the following example using \texttt{SUM}, a nonelemental function:

\begin{verbatim}
REAL A, B
DIMENSION A(10,10), B(10)
WHERE(B .GT. 0.0)  B = SUM(A, DIM=1)
\end{verbatim}

Since \texttt{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:

\begin{verbatim}
REAL A, B, C
DIMENSION A(10,10), B(10), C(10)
WHERE(C .GT. 0.0)  B = SUM(LOG(A), DIM=1)/C
\end{verbatim}

Because \texttt{SUM} is nonelemental, all of its arguments are evaluated fully regardless of whether they are elemental or not. In this example, \texttt{LOG(A)} is fully evaluated for all elements in A even though \texttt{LOG} is elemental. Notice that the mask is applied to the result of the \texttt{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.

\textbf{Compatibility}

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

\textbf{See Also:} FORALL, Arrays

\textbf{Examples}

\begin{verbatim}
REAL(4) a(20)
  . . .
WHERE (a > 0.0)
  a = LOG (a)
  !LOG is invoked only for positive elements
END WHERE
\end{verbatim}

\textbf{WRAPON}

\textbf{Graphics Function:} Controls whether text output with the \texttt{OUTTEXT} function wraps to a new line or is truncated when the text output reaches the edge of the defined text window.

\textbf{Module:} USE DFLIB
Syntax

\[ \text{result} = \text{WRAPON} \ (\text{option}) \]

\textbf{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 \textit{option}.

\textbf{WRAPON} does not affect font routines such as \textbf{OUTGTEXT}.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also: \textbf{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%numx pixels = -1
wc%numy pixels = -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 SETTEXTEXTWINDOW( INT2(1),INT2(1),INT2(5),INT2(25))
CALL SETTEXTPosITION(INT2(1),INT2(1), curpos )
status2 = WRAPON( $GWRAPOFF )
status4 = SETTEXTCOLORRGB(#00FF00)
DO i = 1, 5
    CALL OUTTEXT( 'Here text does wrap. ')
END DO
CALL SETTEXTEXTWINDOW(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
**WRITE** *(eunit, format, [,iostat] [,err]) [io-list]*

**Unformatted**

**WRITE** *(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 *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 **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**

```plaintext
! 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 (Generic):** See **IEOR**.

**Example**

```plaintext
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**

\[
\text{result} = \text{ZEXT} \ (x)
\]

\(x\)

(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**.

<table>
<thead>
<tr>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IZEXT</td>
<td>LOGICAL(1)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td></td>
<td>LOGICAL(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>JZEXT</td>
<td>LOGICAL(1)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>LOGICAL(2)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>LOGICAL(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(1)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(2)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>KZEXT 1</td>
<td>LOGICAL(1)</td>
<td>INTEGER(8)</td>
</tr>
</tbody>
</table>
**Compatibility**

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

**Example**

Consider the following example:

```plaintext
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.

1 Alpha only
Glossary

A - B - C - D - E - F - G - H - I - K - L - M - N - O - P - Q - R - S - T - U - V - W

Glossary A

absolute pathname
On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, a directory path specified in fixed relationship to the root directory. On Tru64 UNIX and Linux systems, the first character is a slash (/). On Windows NT (including Windows 2000) and Windows 9* 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 95/90 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.

---

**Glossary B**

**background process**
On Tru64 UNIX and Linux 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).

### Glossary 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.

**column-major order**
See order of subscript progression.

**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. The following types of complex constants are available on all systems: COMPLEX (COMPLEX(4)) and DOUBLE COMPLEX (COMPLEX(8)). On OpenVMS, Tru64 UNIX, and Linux systems, COMPLEX(16) is also available.

**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 95/90, 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. A
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.

---

**Glossary 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 95/90 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 Compaq 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(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(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(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(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(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 (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux 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.

### Glossary 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 95/90 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.

**Glossary 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 95/90 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 95/90 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 Tru64 UNIX and Linux 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 95/90, 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](#).

**Glossary G**

**generic identifier**
A 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.

Glossary 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. Compaq Fortran 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 nH, 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.

Glossary 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
A 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 95/90 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 95/90 library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available to any Fortran 95/90 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:

\[ \frac{\text{terminal value} - \text{initial value} + \text{increment value}}{\text{increment value}} \]

---

**Glossary 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.

**Glossary 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 Compaq Fortran language element or interpretation that is not part of the Fortran 95 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 95/90 statement; for example, constants, and statement keywords.

**line**

A source form record consisting of 0 or more characters. A standard Fortran 95/90 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 Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* 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 Compaq 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.

## Glossary 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 Tru64 UNIX and Linux systems, an argument to the make command containing a sequence of entries that specify dependencies. On Windows NT (including Windows 2000) and Windows 9* systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependencies. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs.

For more information on makefiles for Tru64 UNIX and Linux systems, see `make(1)`. For more information on makefiles for Windows NT (including Windows 2000) and Windows 9* systems, see *Building Projects with NMAKE* in the *Programmer's Guide*.

**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.

## Glossary 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(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 95/90 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 95/90, 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.
Glossary 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. Also called column-major order.

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.

Glossary 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 95/90 named constant.

**parent window**
A window that has one or more child windows.

**pathname**
On Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux systems, the path from the root directory to a subdirectory or file. See also root.

**pipe**
On Windows NT (including Windows 2000), Windows 9*, Tru64 UNIX, and Linux systems, 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 98 on x86 processors).

**pointer**
Is one of the following:
- A Fortran 95/90 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 Compaq Fortran 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 95/90 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.

---

**Glossary 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.

---

**Glossary 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, Tru64 UNIX, and Linux 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. Compaq Fortran uses these numbered, fixed-length cells to calculate the component's physical position in the file.

**relative pathname**
On Tru64 UNIX, Linux, Windows NT (including Windows 2000), and Windows 9* systems, a directory path expressed in relation to any directory other than the root directory. *Contrast with absolute pathname.*

**root**
On Windows NT (including Windows 2000) and Windows 9* 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 Tru64 UNIX and Linux 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.

### Glossary 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:

\[
y = \text{RAN} \ (i).
\]

**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 95/90, 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 95/90, 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.

**Glossary 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.

Glossary 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.

### Glossary 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.

**Glossary W**

**wait function**
A function that blocks the execution of a calling thread until a specified set of conditions has been satisfied.